Liceo Scientifico Statale "A. Volta"

Corso di Approfondimento di Matematica

Anno Scolastico 2018/19

Introduzione alla programmazione in C++ e applicazioni

Autore:
Leonardo Fiore

Ringraziamenti:
Questo corso, giunto al suo terzo anno di vita, è stato tenuto, prima di me, da Federico Granata, che ne ha curato la prima edizione nell'anno scolastico 2016/17. A lui si devono quindi le principali scelte di indirizzo; inoltre, diversi capitoli di queste dispense sono stati sviluppati a partire dal cospicuo materiale che Federico ha generosamente lasciato a mia disposizione.

21 marzo 2019

Indice

1 Aritmetica

Prima di passare alla programmazione, richiameremo in questo capitolo alcune nozioni fondamentali dell'aritmetica, per arrivare, nel prossimo, ad illustrare come sono rappresentati i numeri al calcolatore.

1 I numeri interi

1.1 Somma e prodotto

L'insieme di tutti i numeri interi dotati di segno viene convenzionalmente indicato con $\mathbb{Z}$:

$$\mathbb{Z} = \{\dots, -3, -2, -1, 0, +1, +2, +3, \dots\}.$$

Su questo insieme sono disponibili le due operazioni fondamentali di somma e prodotto, delle quali riassumiamo qui le principali proprietà.

- La somma è associativa e commutativa. Ammette un elemento neutro (lo zero). Inoltre, ogni numero intero ha un inverso additivo, che si chiama "opposto": l'opposto di n si indica con $-n$ ed è, per definizione, quel numero che sommato con n dà 0 (cioè l'elemento neutro).

- Il prodotto è associativo e commutativo. Ammette un elemento neutro (l'uno). Non esistono però, in generale, i "reciproci", cioè gli inversi moltiplicativi degli elementi (per esempio, 2 non ha un reciproco, poiché nessun numero intero, moltiplicato per 2, dà 1).

Somma e prodotto interagiscono mediante la proprietà distributiva: comunque si scelgano $a, b, c \in \mathbb{Z}$, si ha che

$$a(b + c) = ab + ac.$$

Vale infine la legge di annullamento del prodotto, per la quale un prodotto di numeri interi è 0 se, e solamente se, si annulla uno dei fattori:

$$ab = 0 \iff a = 0 \vee b = 0.$$

L'implicazione $\Longleftarrow$ si può riassumere nell'affermazione "moltiplicando per 0, si ottiene sempre 0" e prende anche il nome di "proprietà assorbente di 0": a ben vedere, è una conseguenza della distributività e del ruolo di 0 come elemento neutro della somma. L'implicazione $\Longrightarrow$ è invece indipendente dalle proprietà già introdotte, e quindi foriera di un'informazione effettivamente nuova circa la natura dei numeri interi, che va sotto il nome di "integrità" di $\mathbb{Z}$.

1.2 La divisione

La divisione tra numeri interi può essere pensata come operazione inversa a quella di prodotto: calcolare $10 : 5$ significa chiedersi quale numero intero q, moltiplicato per 5, dia 10, cioè quali soluzioni intere ammetta l'equazione $5q = 10$. La risposta è, evidentemente, una soltanto: $q = 2$.

La divisione intera, intesa in questo senso, ha un limite evidente: non sempre è possibile eseguirla. $13 : 5$ non ha risultato, perché nessuno numero $q \in \mathbb{Z}$, moltiplicato per 5, dà 13: se si prova con $q = 2$, si ottiene $5q = 5 \cdot 2 = 10$, che è meno di 13; se si prova $q = 3$, si ottiene $5q = 5 \cdot 3 = 15$, che è già troppo. Diversi atteggiamenti sono possibili davanti a questo problema.

Soluzione 1: i numeri razionali

Una strada possibile è ampliare l'insieme $\mathbb{Z}$, costruendo un ambiente più grande in cui esista un numero, compreso tra 2 e 3, che possa essere considerato il risultato della divisione 13:5. È così che da $\mathbb{Z}$ si ottiene l'insieme $\mathbb{Q}$ dei numeri razionali, ove la divisione (per numeri diversi da zero) è sempre possibile. In $\mathbb{Q}$, il risultato di $13 : 5$ è $\frac{13}{5}$, dove $\frac{13}{5}$ è, a ben vedere, una quantità appositamente definita in modo che, moltiplicandola per 5, si ottenga 13. La risposta alla domanda "quanto fa $13 : 5$?", che non è possibile dare in $\mathbb{Z}$, in $\mathbb{Q}$ invece esiste ed è tautologica.

Soluzione 2: rinunciare

Una seconda possibilità è rinunciare a dividere 13 per 5. Si accetta, in altre parole, che per alcune coppie dividendo-divisore la divisione sia possibile, e per altre no. Tale rinuncia non è necessariamente infeconda: per esempio, all'interno di un ambiente numerico come $\mathbb{Z}$ dove non sempre si può dividere un numero per un altro, diventa interessante chiedersi quali numeri siano divisibili per quali altri, cioè studiare la relazione di divisibilità tra numeri.

Per indicare che la divisione $a : b$ è possibile, si scrive in genere $b|a$, che si legge "b divide a", "b è un divisore di a", "a è un multiplo di b". Alla relazione di divisibilità fanno riferimento molte nozioni di largo uso in matematica (massimo comune divisione, minimo comune multiplo, numeri primi, ...): tutte debbono la propria dignità precisamente al fatto che "non sempre si può dividere".

Soluzione 3: la divisione con resto

Un terzo rimedio contro l'impossibilità di calcolare $13 : 5$ consiste nel rimanere dentro all'insieme numerico $\mathbb{Z}$, riconsiderando però in un senso più lato l'operazione di divisione. Quintuplicando un numero intero q, non si otterrà mai 13. Se però a $5q$ ci concediamo di apportare una correzione intera r, allora la speranza di ottenere 13 immediatamente rinasce. Per esempio:

- Se $q = 2$, $5q = 10$, che è troppo poco! Se però al risultato 10 possiamo aggiungere una correzione $r = +3$, allora otterremo 13 come sperato. Diremo allora che $13 : 5$ fa 2 con resto 3.

- Se $q = 3$, $5q = 15$, che è troppo! Se però sul risultato possiamo intervenire con una correzione $r = -2$, allora otterremo 13 come sperato. Diremo che $13 : 5$ fa 3 con resto -2.

Il risultato della divisione $13 : 5$ ha così la forma non più di un singolo intero q tale che $5q = 13$, ma di una coppia (q,r) di quoziente e resto, tale che $5q + r = 13$. Il risultato della divisione ora esiste di certo, ma ha il difetto di non essere univoco:

$$13 : 5 = \begin{cases} \dots \\ q = -2, \quad r = 23 \\ q = -1, \quad r = 18 \\ q = 0, \quad r = 13 \\ q = 1, \quad r = 8 \\ q = 2, \quad r = 3 \\ q = 3, \quad r = -2 \\ q = 4, \quad r = -7 \\ \dots \end{cases}$$

Per recuperare l'unicità, bisogna individuare un criterio che consenta di scegliere una sola coppia (q, r) tra tutte quelle elencate. Una possibilità è questa:

Convezione. *Si chiede che il resto r di una divisione debba essere sempre maggiore o uguale a 0, ma strettamente minore del modulo del divisore.*

Nel nostro caso, deve aversi $0 \leq r < 5$, dunque possiamo dire che l'unico risultato di $13 : 5$ è $(q = 2, r = 3)$. La divisione intera con resto ammette sempre, purché il divisore sia diverso da zero, uno ed un solo risultato:

Teorema/definizione (divisione euclidea)**.** *Dati due numeri $a, b \in \mathbb{Z}$ con $b \neq 0$, esiste una ed una sola coppia di numeri $q \in \mathbb{Z}$, $r \in \mathbb{Z}$, con $0 \leq r < |b|$, tali che $qb + r = a$. Tali numeri q ed r si chiamano rispettivamente quoziente e resto della divisione di a per b.*

1.3 Extra: le classi di resto

Può essere interessante osservare il susseguirsi di resti e quozienti quando si dividono gli interi per un divisore fissato (per esempio, 5). La figura qui sotto esemplifica quello che accade:

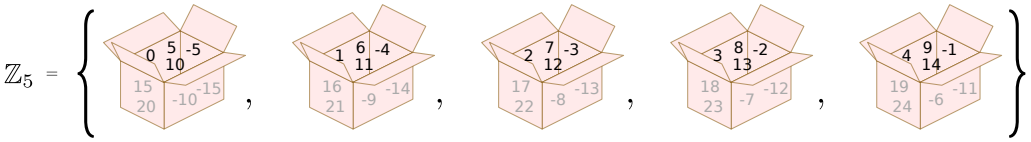
N	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
q	-1	-1	-1	-1	-1	0	0	0	0	0	1	1	1	1	1	2	2
r	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1

I resti (i numeri rossi) variano tra un numero solo finito di possibilità, e si ripetono ciclicamente. Vogliamo ora concentrare la nostra attenzione esclusivamente sui resti; immaginiamo, quindi, di pensare ai numeri interi avendo riguardo soltanto per il loro resto nella divisione per 5. Vedremo come questo punto di vista ci condurrà verso un ambiente numerico del nuovo: d'ora innanzi, ci riferiremo ad esso col termine "aritmetica delle classi di resto modulo 5" e, lo indicheremo con il simbolo $\mathbb{Z}_5$.

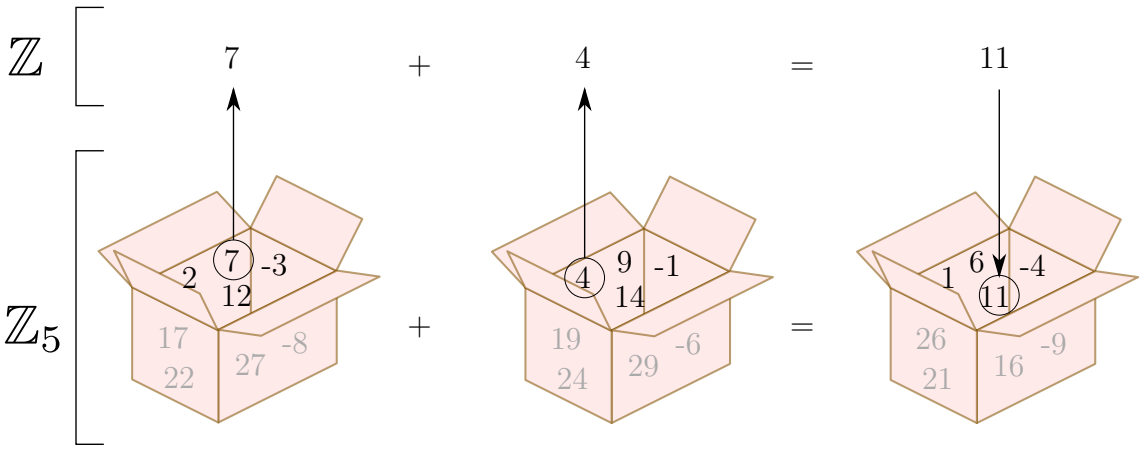
Cominciamo con l'osservare che:

- il -10, il -5, lo 0, il 5, il 10, ecc., avendo tutti uno stesso resto (zero) nella divisione per 5, divengono indistinguibili l'uno dall'altro in $\mathbb{Z}_5$, e finiscono così per essere nomi diversi di uno stesso numero;
- un discorso analogo si può fare per i numeri -9,-4, 1, 6, 11, 16, ecc.: tutti hanno lo stesso reso (uno) nella divisione per 5, e diventano quindi, in $\mathbb{Z}_5$, uno stesso, singolo numero;
- e via dicendo...

Insomma, $\mathbb{Z}_5$ consiste solamente di cinque numeri, ciascuno dei quali può essere indicato con molti nomi equivalenti. Ognuno di questi cinque numeri può essere pensato come una "scatola" (più formalmente, una classe di equivalenza) di numeri interi:



Per introdurre un nuovo ambiente numerico non è però sufficiente indicarne gli elementi: occorre definire anche le operazioni tra di essi. Dobbiamo dunque domandarci, per esempio, come calcolare la somma di due numeri di $\mathbb{Z}_5$, cioè due "scatole". L'idea è molto semplice: si "pescano" due **rappresentanti** $a \in \mathbb{Z}$ e $b \in \mathbb{Z}$ scelti a caso, rispettivamente, dalla prima e dalla seconda scatola; se ne determina la somma $a + b \in \mathbb{Z}$ (come numeri interi); si seleziona la scatola in cui $a + b$ ricade:



Si osservi che la procedura appena proposta per il calcolo della somma richiede di compiere una scelta arbitraria: perché la "somma di due scatole" risulti una nozione ben definita, è quindi indispensabile verificare che, variando la scelta dei rappresentanti, la scatola che il nostro procedimento produce come risultato non cambi. Così è effettivamente, anche se ometteremo di dimostrarlo.

Disponiamo quindi di una ben definita nozione di somma in $\mathbb{Z}_5$. Con un metodo non dissimile si può introdurre in $\mathbb{Z}_5$ pure una ben definita nozione di prodotto. Somma e prodotto in ereditano, inoltre, molte delle proprietà di cui godono in $\mathbb{Z}$.

1.4 Esercizi

1. Verificare o confutare:
   a) 27 diviso 7 fa 3 con resto 6
   b) 27 diviso 7 fa 4 con resto -1
2. Eseguire le seguenti divisioni con resto:
   a) 27 diviso -7
   b) -27 diviso 7
3. Dire se le seguenti affermazioni sono vere o false:
   a) $-3 = 3$ in $\mathbb{Z}_6$
   b) $7 = 1$ in $\mathbb{Z}_4$
4. Scrivere la tavola di composizione per l'operazione di prodotto in $\mathbb{Z}_5$ (cioè calcolare tutti i prodotti possibili degli elementi, raccogliendoli in una tabella a doppia entrata). Fare lo stesso per il prodotto in $\mathbb{Z}_6$.
5. Nella sezione 1.1 si è osservato che, in $\mathbb{Z}$, non tutti gli elementi ammettono un inverso moltiplicativo. Riflettendo su questa affermazione, si risponda alle seguenti domande:
   a) quali sono gli elementi invertibili in $\mathbb{Z}$?
   b) quali sono gli elementi invertibili in $\mathbb{Z}_5$?
   c) quali sono gli elementi invertibili in $\mathbb{Z}_6$?
   d) quali sono gli elementi invertibili in $\mathbb{Q}$?
6. In $\mathbb{Z}_5$ vale la legge di annullamento del prodotto? E in $\mathbb{Z}_6$? Individuare gli eventuali zero-divisori.
7. Calcolare, rapidamente, quanto fa $2657819 \cdot 21 + 7$ in $\mathbb{Z}_3$.
8. Il piccolo teorema di Fermat afferma che, se p è un numero naturale primo, e a è un qualunque numero intero, allora a^p (che significa a moltiplicato per sé stesso p volte) coincide con a in $\mathbb{Z}_p$. Verificare che il teorema vale se $p = 5$.

2 Sistemi numerici

2.1 I sistemi posizionali

Il sistema numerico correntemente utilizzato oggi in matematica si chiama "sistema posizionale in base 10", o più semplicemente "sistema decimale". L'alfabeto di riferimento è costituito da dieci simboli, le cifre, che consentono di indicare i primi dieci numeri naturali (dallo zero al nove). Per rappresentare numeri naturali più grandi,

occorrerà giustapporre più cifre, che faranno riferimento ad ordini di grandezza decimali via via crescenti (unità, decine, centinaia, ...) a seconda della loro posizione. Così, 476 significa 4 centinaia, 7 decine e 6 unità, cioè:

$$476 := 6 \cdot 10^0 + 7 \cdot 10^1 + 4 \cdot 10^2.$$

In generale, il numero di $n+1$ cifre che si scrive come $a_n ... a_0$ (essendo ciascuno degli a_i una cifra tra 0 e 9) vale:

$$a_0 \cdot 10^0 + a_1 \cdot 10^1 + \cdots + a_n \cdot 10^n = \sum_{i=0}^{n} a_i \cdot 10^i.$$

Il sistema decimale fornisce un linguaggio completo (è possibile significare ogni numero naturale) e irridondante (c'è un unico modo possibile di scrivere un dato numero naturale):

Teorema (Esistenza ed unicità della rappresentazione decimale)**.** *Ogni numero naturale $a \in \mathbb{N}$ ha una ed una sola rappresentazione decimale, cioè esiste una ed una sola successione finita di cifre $a_i \in \{0, \dots 9\}$ tali che $a = \sum_{i=0}^{n} a_i \cdot 10^i$.*

Si osservi che il teorema è vero grazie al fatto che 10 è, allo stesso tempo, il **numero di cifre** impiegate e la **base** di riferimento per il calcolo degli ordini di grandezza:

- se, mantenendo la base uguale a 10, utilizzassimo 11 cifre ($1, 2, 3, 4, 5, 6, 7, 8, 9, A$, dove A vale 10), allora il linguaggio diventerebbe ridondante: per significare cinquanta, potremmo scrivere equivalentemente 50 (cioè cinque decine, e zero unità: $50 = 5 \cdot 10^1 + 0 \cdot 10^0$) oppure 4A (cioè quattro decine, e dieci unità: $50 = 4 \cdot 10^1 + 10 \cdot 10^0$).

- se, mantenendo la base uguale a 10, utilizzassimo solo 9 cifre ($1, 2, 3, 4, 5, 6, 7, 8$, privandoci del 9), allora il linguaggio risulterebbe incompleto: nessuna sequenza di cifre potrebbe rappresentare il nove, o il diciannove, o il centoventinove,...

Se però cambiamo in modo corrispondente il numero di cifre impiegate e la base, allora possiamo ottenere nuovi sistemi posizionali di numerazione, con le stesse desiderabili proprietà di completezza e di irridonandanza del sistema decimale.

Teorema (Esistenza ed unicità della rappresentazione posizionale in base b)**.** *Fissata una base $b \geq 2$, ogni numero naturale $a \in \mathbb{N}$ ha una ed una sola rappresentazione in base b, cioè esiste una ed una sola successione finita di cifre $a_i \in \{0, \dots b-1\}$ tali che $a = \sum_{i=0}^{n} a_i \cdot b^i$.*

La base in cui una data succesione di cifre deve essere interpretata, quando diversa da 10, sarà indicata a pedice. Così, per esempio, 10011_2 è quel numero naturale la cui rappresentazione binaria (cioè in base 2) consta delle cifre $a_4 = 1$, $a_3 = 0$, $a_2 = 0$, $a_1 = 1$, $a_0 = 1$. Si ha dunque:

$$10011_2 = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 = 16 + 2 + 1 = 19.$$

La rappresentazione posizionale in base b può essere estesa anche alla rappresentazione di numeri non interi, a patto di impiegare cifre "dopo la virgola", che corrispondano ad ordini negativi di grandezza (cioè decimi, centesimi, millesimi, ecc. in base 10; mezzi, quarti, ottavi, ecc. in base 2; ...). Per esempio:

$$0,1001_2 = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = 0,5 + 0,0625 = 0,5625.$$

Quale che sia la base b, ammettendo la presenza di cifre oltre la virgola (eventualmente infinite), si riescono a rappresentare tutti i numeri reali: detto altrimenti, la rappresentazione posizionale in base b, generalizzata in modo da ammettere la presenza di cifre oltre la virgola, è un linguaggio completo per rappresentare i numeri reali (esiste tuttavia, in quest'ambito, un piccolo problema di ridondanza: in base 10, per esempio, 1 e $0,\overline{9}$ sono due rappresentazioni alternative del numero 1).

Come mostrano i due esempi sopra esposti, non è difficile ottenere la rappresentazione in base 10 un numero scritto in una base b qualsiasi. Più complesso può essere il passaggio inverso, cioè ottenere la rappresentazione in una base b assegnata di un intero fornito in notazione decimale.

2.2 Conversione da base 10 a base b

Immaginiamo di voler scriverei il numero 37 in base 2. In altre parole, vogliamo determinare cifre binarie $a_0, \ldots, a_n$ (ciascuna delle quali può avere solo due valori: 0 o 1) tali che:

$$37 = a_0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + a_3 \cdot 2^3 + \ldots$$

Ci troviamo così di fronte ad una sorta di equazione da risolvere, in cui le incognite sono le cifre $a_0, \ldots, a_n$. Procediamo dividendo entrambi i membri per 2:

- Dividendo la somma scritta al termine di destra per 2, otteniamo:
  - come resto, $r = a_0$;
  - come quoziente, $q = a_1 + a_2 \cdot 2^1 + a_3 \cdot 2^2 + \ldots$: cioè la somma di partenza, in cui il termine di testa a_0 è stato eliminato e tutti gli altri sono stati retrocessi di un ordine di grandezza.
- Diviendo 37 per 2, otteniamo:
  - come resto, 1;
  - come quoziente, 18.

Possiamo quindi concludere che:

$$a_0 = 1 \qquad 18 = a_1 + a_2 \cdot 2^1 + a_3 \cdot 2^2 + \ldots$$

Abbiamo così determinato una cifra - a_0 - e ottenuto una nuova equazione, dello stesso aspetto di quella di partenza, ma meno complessa, alla quale applicheremo un trattamento identico a quello appena utilizzato:

- Dividendo per 2 la somma di destra, otterremo questa volta
  - come resto, $r' = a_1$

  - come quoziente, $q' = a_2 + a_3 \cdot 2^1 + \ldots$: cioè la somma precedente, privata del termine di testa a_1 e con gli altri termini retrocessi di un ulteriore ordine di grandezza.

- D'altra parte, se dividiamo 18 per 2 otteniamo:
  - come resto, $r' = 0$
  - come quoziente, $q' = 9$

Possiamo quindi esser certi che:

$$a_1 = 0 \qquad 9 = a_2 + a_3 \cdot 2^1 + \ldots$$

Iterando il procedimento quante volte necessario, arriveremo a determinare tutte le cifre binarie del numero 37.

Riepilogando, per ottenere la rappresentazione binaria di un numero intero, dobbiamo procedere come segue:

- dividiamo il numero per 2, e salviamo il quoziente e il resto (quest'ultimo potrà essere soltanto 0 oppure 1);
- dividiamo il quoziente dell'operazione precedente per 2, e salviamo nuovamente il quoziente e il resto;
- ripetiamo questa procedura fino ad ottenere 0 come quoziente;
- leggendo la sequenza dei resti dal fondo verso la cima, troviamo la rappresentazione binaria cercata.

Nel caso del numero 37, l'algoritmo procede in questo modo:

			Quoziente	Resto
37	:	2	18	1
18	:	2	9	0
9	:	2	4	1
4	:	2	2	0
2	:	2	1	0
1	:	2	**0**	1

La nostra esecuzione termina, perché abbiamo ottenuto come quoziente il numero 0: leggendo ora dal basso verso l'altro la sequenza di resti troviamo 100101_2, che è la rappresentazione binaria desiderata.

Con un procedimento analogo possiamo scrivere in binario anche numeri non interi. La parte intera di un numero come 37,75 si converte come abbiamo appena descritto, mentre per la parte frazionaria (cioè 0,75) si applica il seguente algoritmo:

- moltiplichiamo per 2 la parte frazionaria, e salviamo parte frazionaria e intera del risultato (quest'ultima potrà essere soltanto 0 o 1);

- moltiplichiamo la parte frazionaria ottenuta al passo precedente per 2, e salviamo nuovamente parte frazionaria e intera del risultato;
- ripetiamo questa procedura quante volte necessario;
- quello che troviamo leggendo da cima a fondo tutte le parti intere successivamente ottenute è lo sviluppo binario cercato (che potrà risultare finito o periodico).

Vediamo qualche esempio: iniziamo dalla conversione del numero 37,75.

		Risultato	Parte intera	Parte frazionaria
0,75	× 2	1,5	1	0,5
0,5	× 2	1,0	1	**0**

Leggendo dall'alto verso il basso le parti intere troviamo quindi che la rappresentazione binaria di $0,75$ è $0,11_2$ (la procedura è stata interrotta nel momento in cui si è ottenuto 0 come parte frazionaria: prolungarla oltre avrebbe prodotto solo cifre nulle non significative). La rappresentazione binaria di 37,75 si otterrà a questo punto riaffiancando, scritte in binario, la parte intera e quella frazionaria del numero:

$$37,75_{10} = 100101,11_2.$$

Proviamo ora a convertire 37,6: per la parte frazionaria avremo

		Risultato	Parte intera	Parte frazionaria
0,6	× 2	1,2	1	0,2
0,2	× 2	0,4	0	0,4
0,4	× 2	0,8	0	0,8
0,8	× 2	1,6	1	**0,6**
...	...	...	...	...

Abbiamo trovato di nuovo $0,6$, ossia il numero da cui siamo partiti: questo significa che la rappresentazione binaria della parte decimale è periodica:

$$37,6_{10} = 100101,\overline{1001}_2.$$

2.3 Esercizi

1. In base 16, si utilizzano come simboli per le cifre $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$. Scrivere in notazione decimale DE_{16}, $A0_{16}$, 10_{16}.
2. Convertire in base 10 i numeri 14_5, 1010_2.
3. Scrivere i numeri interi da 0 fino a 16 in binario.
4. Scrivere i numeri interi da 0 fino a 27 in base 3.
5. Come si scrivono, in base 7, i numeri 7^{100} e $7^{100} - 1$?

6. Convertire 27 e 15 in binario.
7. Convertire $1110010101,11001_2$ e $10111,\overline{1}_2$ in base 10.
8. Convertire $142,4$ e $87,15$ in binario.

2 Rappresentare l'informazione

Per memorizzare una qualunque informazione (un numero, un testo, un brano musicale), un computer riserva un numero prefissato di caselle di memoria (bit), ciascuna delle quali può ospitare una cifra 0 o una cifra 1: detto altrimenti, le "parole" del linguaggio utilizzato dal computer sono sequenze di cifre binarie aventi una lunghezza predeterminata. Per convertire l'informazione da memorizzare in una sequenza di cifre binarie, si rende necessario un procedimento che va sotto il nome di *codifica*, da pensarsi come una traduzione basata su un sistema di regole convenzionali, che varierà a seconda del tipo di informazione da trattare.

1 Rappresentazione dei numeri naturali

Per codificare un numero naturale, il computer utilizza semplicemente la successione delle sue cifre binarie. Riservando, per esempio, n bit di memoria, si potranno scrivere i primi 2^n numeri naturali, da 0 (che sarà una successione di n zeri) fino a $2^n - 1$ (che sarà una successione di n uni). Il numero di bit più comunemente riservato per la rappresentazione di un numero naturale è $n = 32$; è così possibile memorizzare ogni numero naturale compreso tra 0 e $2^{32} - 1 = 4\ 294\ 967\ 295$.

2 Rappresentazione dei numeri interi

Per rappresentare un numero intero è necessario ricorrere a un sistema di codifica più raffinato, che consenta di tenere conto del segno. Delle diverse possibilità che esistono proponiamo, qui di seguito, le più significative.

2.1 Modulo e segno

La soluzione più intuitiva per la rappresentazione degli interi consiste nel riservare un singolo bit al segno e i restanti al valore assoluto. Se, per esempio, immaginiamo di avere a disposizione $n = 8$ bit di memoria, scriveremo nel primo uno 0 oppure un 1 a seconda che il numero da codificare sia, rispettivamente, positivo o negativo; nei successivi 7 bit riporteremo invece le cifre binarie del modulo (che potrà quindi variare tra 0 e $2^{n-1} - 1 = 127$). Saremo così riusciti a rappresentare tutti gli interi dell'intervallo $\{-(2^{n-1} - 1), \ldots, 2^{n-1} - 1\} = \{-127, \ldots, +127\}$. Un inconveniente di questa codifica è la presenza di due modi diversi per rappresentare lo zero (0 può infatti leggersi equivalentemente come $+0$, che si rappresenta come una successione di tutti zeri, ma anche come -0, che si rappresenta scrivendo 1 nel bit del segno e 0 nei rimanenti).

2.2 Slittamento

Un'altra soluzione piuttosto semplice per rappresentare i numeri interi provvisti di segno è ricondursi, mediante traslazione, a numeri naturali, per i quali già abbiamo una codifica. Avendo a disposizione, per esempio, 8 bit, possiamo rappresentare gli interi dell'intervallo $\{-2^7, \dots 2^7 - 1\} = \{-128, \dots, +127\}$ (che è un intervallo di $2^8 = 256$ numeri interi quasi centrato sullo zero, con uno sbilanciamento unitario a sinistra) secondo questa regola: all'intero x che si vuol rappresentare, si somma $+2^7$; ne risulta un numero naturale dell'intervallo $\{0, \dots, 2^8 - 1\} = \{0, \dots, 255\}$, che possiamo rappresentare semplicemente scrivendone le cifre binarie negli 8 bit a disposizione:

$$\{-128, \dots, +127\} \underset{-2^7}{\overset{+2^7}{\rightleftharpoons}} \{0, \dots, 255\} \underset{\text{Decod. naturali 8bit}}{\overset{\text{Codifica naturali 8bit}}{\rightleftharpoons}} \left\{ \begin{array}{c} \text{Sequenze di} \\ \text{8 zeri/uni} \end{array} \right\}.$$

Si può ovviamente optare per un intervallo di numeri rappresentabili sbilanciato a destra al posto che a sinistra, a patto di applicare una traslazione non più di 2^7, ma di $2^7 - 1$:

$$\{-127, \dots, +128\} \underset{-(2^7-1)}{\overset{+(2^7-1)}{\rightleftharpoons}} \{0, \dots, 255\} \underset{\text{Decod. naturali 8bit}}{\overset{\text{Codifica naturali 8bit}}{\rightleftharpoons}} \left\{ \begin{array}{c} \text{Sequenze di} \\ \text{8 zeri/uni} \end{array} \right\}.$$

2.3 Extra: complemento a 2

Una tecnica meno elementare, ma più efficace per rappresentare gli interi con segno va sotto il nome di *complemento a 2*. Immaginiamo di voler rappresentare, usando $n = 2^8$ bit, i 256 (=2^8) numeri interi che vanno da -128 fino a $+127$. Come prima, l'idea è di trovare una corrispondenza biunivoca:

$$\{-128, \dots, +127\} \Longleftrightarrow \{0, \dots, 255\},$$

perché sia possibile ricondursi alla codifica dei numeri naturali $\{0, \dots, 255\}$, della quale già disponiamo. La corrispondenza tra i due intervalli non sarà questa volta però data dalla traslazione, ma da un più raffinato abbinamento dei numeri dei due intervalli, che ci accingiamo ora a descrivere.

Torniamo, per un momento, a considerare l'insieme numerico $\mathbb{Z}_5$ discusso nello scorso capitolo, che ricordiamo essere costituito da cinque classi di equivalenza di numeri interi; ricordiamo che a ciascuna di tali classi ci si può riferire mediante uno qualunque dei suoi rappresentanti (cioè mediante uno qualunque degli interi che contiene).

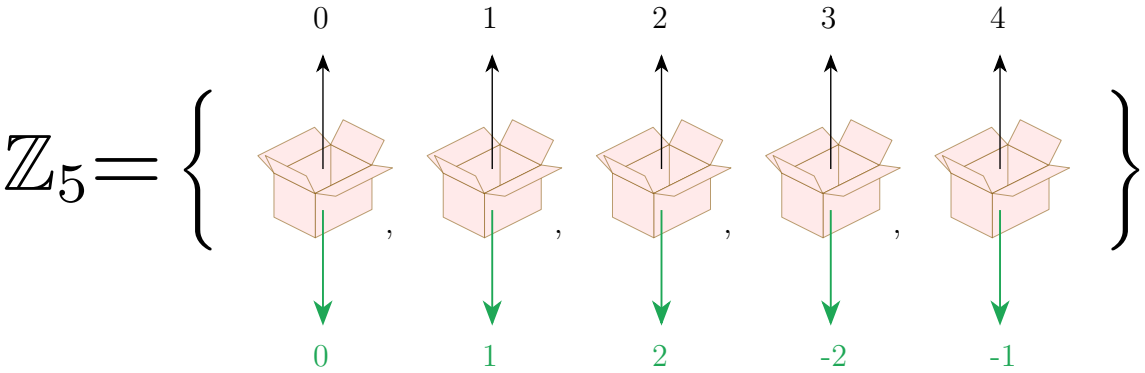
Così, possiamo dire:

$$\mathbb{Z}_5 = \{\text{Classe di } 0,\ \text{Classe di } 1,\ \text{Classe di } 2,\ \text{Classe di } 3,\ \text{Classe di } 4\}$$

ma nulla vieta di riferirsi alla classe di 4 anche chiamandola "classe di -1" e alla classe di 3 chiamandola "classe di -2":

$$\mathbb{Z}_5 = \{\text{Classe di } -2,\ \text{Classe di } -1,\ \text{Classe di } 0,\ \text{Classe di } 1,\ \text{Classe di } 2\}$$

I cinque numeri $-2, -1, 0, 1, 2$ e i cinque numeri $0, 1, 2, 3, 4$ forniscono due modi alternativi di designare le cinque classi di resto modulo 5; in altre parole, sono due *sistemi di rappresentanti* per le classi di resto modulo 5.



Tra due sistemi di rappresentanti esiste sempre un'ovvia corrispondenza biiettiva, che si ottiene facendo corrispondere i rappresentanti di una stessa classe (nell'esempio considerato, $0 \leftrightarrow 0$, $1 \leftrightarrow 1$, $2 \leftrightarrow 2$, $-2 \leftrightarrow 3$, $-1 \leftrightarrow 4$).

Non è difficile convincersi che ogni intervallo di cinque numeri interi consecutivi costituisce un sistema di rappresentanti per $\mathbb{Z}_5$; possiamo così dire che tra due intervalli di cinque interi sono canonicamente determinabili due corrispondenze biunivoche (che sono, in generale, differenti):

- quella che fa corrispondere un intervallo all'altro per traslazione;
- quella che fa corrispondere i due intervalli in quanto sistemi di rappresentanti per le classi di resto modulo 5.

In modo analogo, se torniamo a considerare i due intervalli di 256 numeri interi da cui eravamo partiti

$$\{-128, \ldots, +127\} \qquad e \qquad \{0, \ldots, 255\},$$

sappiamo ora riconoscere, tra di essi, due naturali corrispondenze biunivoche: da un lato, quella realizzata dalla traslazione di 2^7; dall'altro, quella che interpreta i due intervalli numerici come sistemi alternativi di rappresentanti per le classi di resto modulo 256. Se la prima biiezione consente di definire la codifica per slittamento, la seconda è proprio quella che dà vita alla rappresentazione degli interi secondo il metodo del complemento a 2.

3 La rappresentazione dei numeri reali

Un numero reale diverso da 0 si può rappresentare in modo (essenzialmente) unico secondo la cosiddetta *notazione scientifica*, che si ottiene lasciando una sola cifra non nulla dinnanzi alla virgola, e moltiplicando per un'adeguata potenza di 10. Così, per esempio:

$$52,7 = 5,27 \cdot 10^1 \qquad 0,00815 = 8,15 \cdot 10^{-3} \qquad 1,45 = 1,45 \cdot 10^0$$

Un numero scritto in notazione scientifica si presenta dunque nella forma:

$$\pm m_0, m_{-1}m_{-2}m_{-3}\ldots \cdot 10^{\text{esponente}} \qquad \text{con } m_j \in \{0, \ldots, 9\}\ \forall j \text{ e } m_0 \neq 0$$

in cui figurano tre elementi: il *segno*, l'*esponente* e la *mantissa* $m_0, m_{-1}m_{-2}m_{-3}\ldots$.

Anche in base 2 si può parlare di notazione scientifica: tutto funziona, mutatis mutandis, come in base 10, con la particolarità che la cifra delle unità della mantissa m_0, dovendo essere diversa da 0, sarà semplicemente sempre uguale ad 1, e potremo dunque scrivere il numero come:

$$\pm 1, m_{-1}\, m_{-2}\, m_{-3} \ldots \cdot 2^{\text{esponente}} \qquad \text{con } m_j \in \{0, 1\}\ \forall j.$$

È proprio in notazione scientifica binaria che i numeri reali vengono in genere rappresentati al calcolatore. Immaginando di avere a disposizione $n = 32$ bit, li utilizzeremo in questo modo:

- 1 bit per memorizzare il segno del numero: varrà 0 se il numero da rappresentare è positivo, 1 se è negativo.
- 8 bit per l'esponente, che sarà un numero intero, trascritto secondo la codifica per slittamento: useremo, in particolare, una traslazione di $2^7 - 1$ e potremo così rappresentare gli esponenti da -127 fino a $+128$.
- I rimanenti 23 bit saranno dedicati alle cifre $m_{-1}, m_{-2}, \ldots, m_{-23}$ della mantissa (non occorre memorizzare m_0 in quanto è sempre uguale a 1).

Sono in realtà operativamente disponibili sono gli esponenti da -126 fino a $+127$: i due esponenti estremi -127 e $+128$ sono infatti riservati per funzioni particolari:

	Esponente	Cifre $m_{-1}, m_{-2}, \ldots$ della mantissa
Zeri	-127	Tutte nulle
Numeri subnormalizzati	-127	Non tutte nulle
Infiniti	$+128$	Tutte nulle
NaN	$+128$	Non tutte nulle

Guardando alla tabella, notiamo che la nostra codifica rende possibile, modificando il bit riservato al segno, rappresentare due diversi zeri ($+0$ e -0) e due diversi infiniti ($+\infty$ e $-\infty$). Non discuteremo in dettaglio dei numeri subnormalizzati, la cui funzione è quella di rendere meno brusco il salto tra il minimo numero reale positivo rappresentabile (che è 2^{-126}) e lo zero. Infine, NaN sta per "Not a Number", valore che viene talvolta restituito quando si tenta di eseguire operazioni dal risultato indeterminato (per esempio una divisione 0/0 o la radice quadrata di un numero negativo).

La codifica a 32 bit appena descritta per i numeri reali prende il nome di *binary32*, o codifica in *precisione singola*. L'uso di un numero più elevato di bit consente di ampliare l'intervallo di esponenti rappresentabili e di memorizzare un numero maggiore di cifre della mantissa: è il caso degli standard *binary64* e *binary128* descritti nella tabella qui sotto.

Codifica	Bit totali	Segno	Esponente	Mantissa
Precisione singola (binary32)	32	1	8	23
Precisione doppia (binary64)	64	1	11	52
Precisione quadrupla (binary128)	128	1	15	112

3.1 Esempio

Proponiamo ora un esempio di come si rappresenta un numero reale secondo la codifica binary32: rappresenteremo il numero 34,5.

Dobbiamo innanzitutto convertirlo in binario: la parte intera, convertita, dà 100010_2, mentre la parte frazionaria dà $0,1_2$. Complessivamente, quindi,

$$34,5_{10} = 100010,1_2 = 1,000101 \cdot 2^5,$$

dove ci siamo già premurati di scrivere il numero in notazione scientifica (cioè con una sola cifra non nulla, $m_0 = 1$, davanti alla virgola). Possiamo allora riconoscere:

- Il segno: positivo $\rightarrow$ il bit del segno sarà 0.
- L'esponente: $5 \rightarrow$ va rappresentato in uno spazio di 8bit per slittamento, applicando una traslazione di $2^7 - 1$: dobbiamo dunque scrivere, in binario $5 + 127 = 132 = 10000100_2$.
- La mantissa: $1,000101$ (che andrà trascritta omettendo l'uno davanti alla virgola).

La rappresentazione del numero desiderato si otterrà quindi accostando i bit relativi a segno, esponente e mantissa:

$$34,5_{10} = |0|10000100|00010100000000000000000|_{\text{binary32}}.$$

4 I limiti dell'aritmetica del calcolatore

Per ingegnose che possano essere le codifiche adoperate per rappresentare i numeri (naturali, interi, reali, ...), il numero limitato e predeterminato di bit a disposizione rende impossibile ottenere una riproduzione fedele degli insiemi numerici, naturalmente infiniti, che si usano in matematica. L'aritmetica del calcolatore potrà così offrire soltanto un'immagine parziale e deformata dell'aritmetica "umana"; per sottolineare questa differenza, si utilizza il termine *numeri macchina* o *numeri rappresentabili*, in luogo del termine *numeri*, per riferirsi ai numeri così come rappresentati e trattati dal calcolatore secondo una data codifica.

4.1 Overflow nell'aritmetica intera

Si consideri, a titolo d'esempio, la codifica per complemento a 2 a 32 bit degli interi. I numeri macchina sono, in questo caso, il sottoinsieme finito $S = \{-2^{31}, \ldots, 2^{31} - 1\} \subseteq \mathbb{Z}$ (dove 2^{31} vale poco più di 2 miliardi). Questo sottoinsieme S dei numeri rappresentabili (come d'altronde ogni sottoinsieme finito non banale di $\mathbb{Z}$) soffre dei seguenti difetti:

- non coincide con tutto $\mathbb{Z}$;
- non è chiuso rispetto alle operazioni di $\mathbb{Z}$, nel senso che sommando e moltiplicando gli uni con gli altri elementi di S è possibile che si cada fuori da S.

Il secondo dei problemi elencati è senza dubbio il più subdolo: se anche i dati da cui si parte sono numeri perfettamente rappresentabili, non è garantito a priori che siano rappresentabili i risultati delle operazioni eseguiti su di essi. Nella codifica che abbiamo preso come esempio, 2 miliardi e 1 miliardo sono rappresentabili, ma sommandoli il calcolatore non potrà ottenere 3 miliardi, per la ragione che tale numero, dal punto di vista di esso, non esiste affatto. Il risultato effettivamente restituito dal calcolatore per una simile somma dipende dai dettagli di implementazione dell'algoritmo che esegue il calcolo; in ogni caso, sarà un risultato sbagliato. Un siffatto superamento del "limite estremo" dei numeri rappresentabili si dice *overflow*.

4.2 Overflow, underflow e arrotondamento nell'aritmetica reale

Anche nel caso dell'aritmetica reale, vale l'osservazione che il sottoinsieme $T \subseteq \mathbb{R}$ dei numeri rappresentabili secondo una qualunque codifica (precisione singola, precisione doppia, ...) non coincide con tutto $\mathbb{R}$, e non è chiuso rispetto alle operazioni.

Si prenda, come esempio, la codifica in precisione singola. La finitezza dell'insieme dei numeri macchina si manifesta in due distinti effetti.

- Solo un numero finito di ordini di grandezza possono essere coperti dalla codifica: numeri con esponenti troppo piccoli saranno confusi con lo zero (si parla di *underflow*), mentre numeri con esponenti troppo grandi non potranno essere distinti dall'infinito (si parla di *overflow*).

- Le cifre binarie della mantissa che si possono memorizzare sono solo un numero finito (23): la codifica di un numero reale comporta così, in generale, un troncamento della mantissa, vale a dire un *errore di arrotondamento*. Per esempio, il numero reale $3,2$, che si scrive in notazione scientifica binaria come $1,1\overline{0011}\cdot 2^1$, non è un numero macchina; nel rappresentarlo, verranno mantenute solo le prime 23 cifre oltre la virgola della sua mantissa, sicché il computer memorizzerà in realtà $1,10011001100110011001100 \cdot 2^1$, cioè $3,19999980926513671875$ (oppure, approssimando per eccesso, $1,10011001100110011001101 \cdot 2^1$, cioè $3,20000047683715820312 5$); con un errore, in entrambi i casi, dell'ordine dello $0,000001\%$.

Tanto nel caso dei numeri interi, quanto nel caso dei numeri reali, esistono dei limiti estremi da non valicare, oltre i quali la rappresentabilità viene completamente meno: nel caso degli interi, c'è un minimo e un massimo numero rappresentabile; nel caso dei reali, c'è un minimo e un massimo ordine di grandezza che la codifica può raggiungere.

Nel caso dei numeri interi, però, quando ci si mantenga adeguatamente lontano da tali limiti, tanto che si possa esser sicuri di non valicarli, i risultati delle operazioni è garantito siano esatti. Ben altra è la situazione nel caso dei numeri reali: ogniqualvolta si rappresenta un numero, ogniqualvolta si calcola il risultato di un'operazione, ancorché gli ordini di grandezza rimangano a distanza di sicurezza dal massimo e minimo esponente consentito, si otterranno comunque solo risultati *approssimati*. Gli errori di arrotondamento, anche se piccoli, rischiano di accumularsi operazione dopo operazione, determinando conseguenze possibilmente macroscopiche.

5 Esercizi

1. Individuare l'insieme dei numeri interi rappresentabili nelle seguenti codifiche, quando si abbiano a disposizione $n = 10$ bit:
   - modulo e segno;
   - slittamento di 2^9;
   - slittamento di $2^9 - 1$;
   - (facoltativo) complemento a 2.

   Scrivere come interi a 10 bit, in ciascuna delle codifiche sopra indicate, i numeri 5, 0, -125 e -94.

2. Si determini il numero intero che ciascuna delle seguenti sequenze di 8 cifre binarie rappresenta:

   00000000 10000000 01111111 10110010

   al variare della codifica utilizzata tra le seguenti codifiche ad 8 bit:
   - codifica dei naturali;
   - modulo e segno;
   - slittamento di 2^7;
   - slittamento di $2^7 - 1$;
   - (facoltativo) complemento a 2.

3. Rappresentare i due numeri $1412,2$ e $-33552,8$ prima in precisione singola (binary32) e poi in precisione doppia (binary64).

4. Calcolare il numero reale rappresentato, in precisione singola (binary 32), dalle seguenti sequenza di 32 bit:

   1 00000000 00000000000000000000000
   1 11111111 00000000000000000000000
   0 01111111 11000000000000000000000
   0 10000000 11010000000000000000000
   0 10000000 11010000000000000000001

3 Primi programmi in C++

1 Hello world

- Aprire il programma Pocket C++ (che trovate sul Desktop nella cartella AMAT) e salvare un file bianco, selezionando codice C++ come tipo di file e utilizzando l'estensione .cpp
- Scrivere il seguente programma:

```
#include <iostream>
int main()
{
   std::cout << "Hello world";
   return 0;
}
```

- Compilarlo premendo F9
- Terminata la compilazione, eseguirlo premendo Ctrl+F9

2 Lettura e scrittura di variabili

Si considerino i due programmi seguenti:

```cpp
#include <iostream>
int main(){
   int a;
   a = 5;

   std::cout << (a+1);
   std::cout << "\n";

   std::cout << (a+1);
   std::cout << "\n";

   return 0;
}
```

```cpp
#include <iostream>
int main(){
   int a;
   a = 5;

   a = a+1;
   std::cout << a;
   std::cout << "\n";

   a = a+1;
   std::cout << a;
   std::cout << "\n";

   return 0;
}
```

1. Per ciascuno dei programmi, si dica in quali istruzioni `a` viene impiegata in lettura e in quali istruzioni viene impiegata in scrittura.

2. Quale output viene prodotto?

3 Lettura e scrittura di variabili /2

Si consideri il seguente programma:

```
#include <iostream>
int main(){
   int a;
   int b;

   std::cout << "Inserisci il numero a: ";
   std::cin >> a;

   std::cout << "Inserisci il numero b: ";
   std::cin >> b;

   a = b;
   b = a;

   return 0;
}
```

Si immagini che l'utente lo esegua immettendo, nell'ordine, i numeri 10 e 20:

1. Fai un'ipotesi motivata sul valore di `a` e di `b` al termine dell'esecuzione.
2. Prova a trascrivere ed eseguire il programma, arricchendolo con opportuni comandi di output che ti consentano di verificare la tua ipotesi.

4 Calcolatrice intera

Scrivere un programma che:

- chieda all'utente di immettere da terminale due numeri interi e li salvi dentro altrettante variabili (da dichiararsi come numeri interi a 32bit);
- calcoli somma, differenza, prodotto, quoziente intero e resto della divisione del primo numero per il secondo, immagazzinando i risultati ottenuti in altrettante variabili (che saranno sempre di tipo intero a 32 bit);
- confronti i due numeri immessi mediante l'operatore ==, e salvando in un'opportuna variabile booleana il risultato del confronto;
- scriva sul terminale i risultati ottenuti, ciascuno su una diversa riga, ciascuno preceduto da un'opportuna didascalia.

Una volta scritto il programma, si provi ad eseguirlo immettendo come dati: (a) i numeri 7 e 50; (b) i numeri 50 e 50; (c) i numeri 20mila e 200mila; (d) i numeri 200mila e 30mila.

1. I risultati ottenuti sono corretti?
2. Si ricompili il programma, usando questa volta - per tutte le variabili - il tipo "numero naturale a 32bit" in luogo di "numero intero a 32bit". Si provi a rieseguirlo, immettendo ancora le medesime coppie di numeri. Si ottengono gli stessi risultati? Perché?
3. Si ricompili e si riesegua come sopra il programma, usando rispettivamente:
   - variabili "intere a 64bit" per i due numeri e per i risultati delle operazioni.
   - variabili "intere a 64bit" per i due numeri e variabili "intere a 32bit" per i risultati delle operazioni;
   - variabili "intere a 32bit" per i due numeri e variabili "intere a 64bit" per i risultati delle operazioni;

   Cosa si osserva? Perché?

4.1 Soluzione

Ecco un possibile modo di realizzare il programma (usando l'aritmetica dei numeri interi a 32bit, cioè il tipo `int`):

```
#include <iostream>
int main(){

   //DICHIARAZIONI VARIABILI
   int a, b; //I due numeri immessi da tastiera
   int somma, differenza, prodotto, quoziente, resto;
   bool confronto;

   //IMMISSIONE DATI DA TASTIERA
   std::cout << "Immettere a: ";
   std::cin >> a;
   std::cout << "Immettere b: ";
   std::cin >> b;

   //CALCOLO RISULTATI
   somma = a + b;
   differenza = a - b;
   prodotto = a * b;
   quoziente = a / b;
   resto = a % b;
   confronto = a == b;

   //SCRITTURA RISULTATI SU TERMINALE
   std::cout << "Somma a+b: " << somma << "\n";
   std::cout << "Differenza a-b: " << differenza << "\n";
   std::cout << "Prodotto a*b: " << prodotto << "\n";
   std::cout << "Quoziente a/b: " << quoziente << "\n";
   std::cout << "Resto a%b: " << resto << "\n";
   std::cout << "Confronto tra a e b: " << confronto << "\n";

   return 0;
}
```

- Se si immettono i numeri 7 e 50, oppure 50 e 50, il programma fornisce all'utente i risultati corretti per tutte le operazioni.
- Immettendo 20mila e 200mila, o addirittura 200mila e 30mila, il prodotto calcolato dal programma è sbagliato. La ragione è che i due risultati corretti (4milioni e 6milioni) eccedono il massimo intero rappresentabile nella codifica `int`, che è $2^{31} - 1 = 2\,147\,483\,647$, cosicché il calcolo dei due prodotti provoca un *overflow*.

Con i numeri naturali a 32 bit... Se si dichiarano le variabili `a, b, somma, differenza, prodotto, quoziente e resto` di tipo naturale a 32 bit (`unsigned`

`int`) al posto che intero a 32bit (`int`), l'intervallo di rappresentabilità si modifica:

	Minimo	Massimo
Num. rappr. nella codifica degli interi a 32 bit	−2 147 483 648	+2 147 483 647
Num. rappr. nella codifica dei naturali a 32 bit	0	+4 294 967 295

Il calcolo di 20mila∗2000mila non darà più overflow (4miliardi è ora, infatti, un numero rappresentabile); il prodotto 200mila∗30mila continua invece a dare un risultato sbagliato (6miliardi si colloca infatti, anche in questo caso, al di là dell'ultimo intero rappresentabile). Si verificherà inoltre un problema di overflow ogniqualvolta si tenti di calcolare una differenza negativa.

Con i numeri interi a 64... Se si dichiarano le variabili `a`, `b`, `somma`, `differenza`, `prodotto`, `quoziente` e `resto` di tipo intero a 64 bit (`long long`) al posto che intero a 32bit (`int`), l'intervallo di rappresentabilità si amplia notevolmente:

	Minimo	Massimo
Num. rappr. codifica degli interi a 32 bit	−2 147 483 648	+2 147 483 647
Num. rappr. codifica degli interi a 64 bit	−9 223 372 036 854 775 808	+9 223 372 036 854 775 807

Con nessuna delle coppie di numeri interi a e b proposte si verificherà più overflow.

Mischiando interi a 32 e 64bit... Immaginiamo che `a` e `b` siano dichiarate intere a 64 bit, che `prodotto` sia dichiara intera a 32bit, e che l'utente immetta, per esempio, la coppia di numeri 200mila e 30mila: nell'istruzione

```
prodotto = a*b;
```

il prodotto viene calcolato come *prodotto di numeri interi a 64 bit*, e produce dunque il risultato corretto 6miliardi, che viene restituito dall'operatore ∗ come intero a 64 bit. Al momento però di scrivere il risultato dentro alla variabile `prodotto`, interviene una conversione implicita `intero a 64bit` → `intero a 32bit`, che provoca un *overflow* (in quanto 6miliardi non può essere scritto come intero a 32 bit).

Ancora mischiando interi a 32 e 64bit... Immaginiamo ora di trovarci nella situazione speculare, in cui `a` e `b` siano dichiarate intere a 32 bit, mentre `prodotto` sia dichiara di tipo intero a 64bit. Figuriamoci sempre che l'utente immetta, per esempio, la coppia di numeri 200mila e 30mila. Quando viene eseguita l'istruzione:

```
prodotto = a*b;
```

il prodotto viene calcolato come *prodotto di numeri interi a 32 bit*, e produce dunque un risultato di tipo intero 32 bit, che sarà, inevitabilmente, sbagliato a causa dell'*overflow*. Al momento di scrivere tale risultato dentro alla variabile `prodotto`, interviene una conversione implicita `intero a 32bit` → `intero a 64bit`, ma ciò che viene trasformato in un intero a 64bit e salvato dentro a `prodotto` è un risultato sbagliato, che quindi sbagliato rimane.

5 Numeri reali

Si scriva un programma che:

- chiede all'utente due numeri reali (tipo `double`) a e b
- calcola le due seguenti quantità:

$$\frac{1}{a}+\frac{1}{b} \qquad \text{e} \qquad \frac{a+b}{ab}$$

  salvandole in due variabili `ris1` e `ris2` (usare, per tutte le variabili, il tipo double).

- confronta i due risultati mediante l'operatore di eguaglianza ==, e salva il risultato del confronto (che vale `true` se `ris1` e `ris2` hanno il medesimo valore, mentre vale `false` se hanno valori distinti) in un'apposita variabile booleana `confronto`.
- scrive a terminale il valore della variabile `confronto`.

Si formuli una previsione sul valore della variabile `confronto`. Si esegua poi il programma immettendo come dati:

$$25 \text{ e } 1370 \qquad -2 \text{ e } 10 \qquad -25 \text{ e } 10 \qquad -125 \text{ e } 126$$

1. Cosa si osserva?
2. Si arricchisca il programma con opportuni comandi di output, in modo che venga mostrato a terminale, oltre al valore della variabile `confronto`, anche quello delle variabili `ris1` e `ris2`. Cosa si osserva?
3. Si utilizzino i comandi di formattazione dell'output `std::setprecision` e `std::scientific` (dalla libreria `iomanip`) per ottenere che i numeri `ris1` e `ris2` vengano stampati a video in notazione esponenziale con 16 cifre decimali. Cosa si osserva?

5.1 Soluzione

```
#include{iostream}
int main(){
   double a, b, ris1, ris2;
   bool confronto;

   //Immissione di a e b da parte dell'utente
   std::cout << "Inserisci a: ";
   std::cin >> a;
   std::cout << "Inserisci b: ";
   std::cin >> b;

   //Calcoli
   ris1 = 1/a+1/b;
   ris2 = (a+b)/(a*b);
   confronto = (ris1==ris2);

   //Output
   std::cout << "Confronto tra i valori di ris1 e ris2: " << confronto;
   return 0;
}
```

Le due espressioni in questione, $1/a + 1/b$ e $(a+b)/(ab)$, sono chiaramente uguali da un punto di vista matematico: ci aspettiamo dunque che `ris1` e `ris2` abbiano lo stesso valore, che `confronto` valga dunque `true`, e che sul terminale appaia dunque scritto

```
Confronto tra i valori di ris1 e ris2:  1
```

Se si esegue il programma immettendo la prima o la seconda delle quattro coppie di numeri suggerite, cioè $(25, 1370)$ e $(-2, 10)$, si ottiene effettivamente l'output atteso. Se però si tenta con le due rimanenti coppie di valori, cioè $(-25, 10)$ e $(-125, 126)$, il computer, sorprendentemente, restituisce:

```
Confronto tra i valori di ris1 e ris2:  0
```

La ragione per la quale il calcolatore ottiene risultati diversi per due espressioni matematicamente uguali consiste nella natura intrinsecamente *approssimativa* dell'aritmetica reale al calcolatore. Nel capitolo 2 abbiamo osservato come, mentre l'aritmetica intera del computer è esatta (fintanto che non si superino i confini ultimi dell'intervallo di numeri rappresentabili), viceversa la memorizzazione dei numeri reali e le operazioni eseguite tra di essi sono, in generale, gravate da *errori di arrotondamento* (la cui ragione va ricercata nel numero finito e predeterminato dei bit che la codifica riserva alla mantissa). Tali errori si propagano ineluttabilmente lungo i calcoli, cumulandosi gli uni con gli altri in un modo che può dipendere anche sensibilmente dal percorso di calcolo scelto.

Insomma: le due quantità `ris1` e `ris2` che il computer calcola sono solo *approssimazioni* di uno stesso numero reale; e due approssimazioni di uno stesso numero, calcolate per vie diverse, è ben plausibile che differiscano.

La formattazione dell'output Se modifichiamo il programma in modo che riporti a terminale anche i valori di `ris1` e `ris2`:

```
#include<iostream>
int main(){
   //...omissis...
   std::cout << "Confronto tra i valori di ris1 e ris2 " << confronto;
   std::cout << "ris1 vale: " << ris1;
   std::cout << "ris2 vale: " << ris2;
   return 0;
}
```

otterremo come output - se immettiamo, per esempio, la penultima coppia di numeri proposta $(-25, 10)$...

```
 Confronto tra i valori di ris1 e ris2:  0
 ris1 vale:  0.06
 ris2 vale:  0.06
```

Che `confronto` valga 0 indica indiscutibilmente che i valori conservati in memoria per `ris1` e `ris2` sono diversi. Come mai allora troviamo riportati a terminale due numeri identici (0.06) per `ris1` e per `ris2`? La ragione va ricercata nel particolare modo in cui il computer mostra a terminale un numero di tipo `double`: per brevità, ne riporta sempre solamente le prime cifre, cosicché non riproduce fedelmente tutta l'informazione che conserva in memoria. Alcune speciali istruzioni ci consentono però di *regolare* a nostro piacimento queste modalità di formattazione; cosicché, se per esempio scriviamo:

```
#include<iostream>
#include<iomanip>
int main(){
   //... omissis ...
   std::cout << std::scientific;
   std::cout << std::setprecision(16);
   std::cout << "Confronto tra i valori di ris1 e ris2 " << confronto;
   std::cout << "ris1 vale: " << ris1;
   std::cout << "ris2 vale: " << ris2;
   return 0;
}
```

allora il computer riporterà i valori di `ris1` e `ris2` espressi in notazione scientifica decimale, con 16 cifre oltre la virgola:

```
 Confronto tra i valori di ris1 e ris2:  0
 ris1 vale:  6.0000000000000005e-02
 ris2 vale:  5.9999999999999998e-02
```

Possiamo così ora riconoscere la differenza tra i valori delle variabili `ris1` e `ris2` anche nei due numeri stampati a terminale dal computer.

6 Medie pesate

Scrivere un programma che prenda in ingresso tre voti e tre pesi (numeri interi 32bit), e restituisca, come risultato, la media pesata. Qui di seguito viene proposto un esempio:

```
1  #include <iostream>
2  int main(){
3     int voto [3]; //Modo compatto di dichiarare tre variabili
4                   //intere, di nome voto[0], voto[1], voto[2]
5     int peso [3]; //Modo compatto di dichiarare tre variabili
6                   //intere, di nome peso[0], peso[1], peso[2]
7     double media;
8
9     //Inserimento dati da parte dell'utente
10    std::cout << "Inserisci il primo voto: ";
11    std::cin >> voto[0];
12
13    std::cout << "Inserisci il primo peso: ";
14    std::cin >> peso[0];
15
16    //Eccetera...
17
18    //Calcolo della media
19    media = (voto[0]*peso[0] + voto[1]*peso[1] + voto[2]*peso[2])
20            /(peso[0]+peso[1]+peso[2]);
21
22    //Scrittura a terminale
23    std::cout << "Media: ";
24    std::cout << media;
25
26    return 0;
27 }
```

Provare a trascrivere, completandolo, il programma fornito come esempio.

1. Perché non funziona?
2. Sapresti correggerlo?
3. Arricchisci il programma con una variabile boolean `promosso`, il cui valore indichi se la media è sufficiente o meno.

6.1 Soluzione

Correggere il programma Alle righe 19-20 del codice proposto, i due operandi tra cui l'operazione / viene eseguita sono, evidentemente, di tipo `int`, cosicché il compilatore interpreta / come `divisione di due numeri interi` e il risultato restituito dall'operazione è il *quoziente (intero) della divisione euclidea di due numeri interi* - il quale, convertito implicitamente in `double`, viene salvato nella variabile `media`.

Per ottenere un programma correttamente funzionante, la divisione usata per calcolare la media deve essere invece il *quoziente di numeri reali*! Per ottenere questo risultato, abbiamo due possibilità:

- modificare in `double` il tipo delle variabili utilizzate per memorizzare voti e pesi (questo però contrasta con quanto preteso dal testo dell'esercizio, che chiede esplicitamente di usare, per voti e pesi, variabili intere a 32 bit)
- introdurre opportuni operatori di conversione esplicita di tipo nell'espressione che calcola la media:

```
media = ((double)(voto[0]*peso[0]+voto[1]*peso[1]+voto[2]*peso[2]))
        / ((double)(peso[0]+peso[1]+peso[2]));
```

  (sarebbe sufficiente introdurre una conversione esplicita anche su uno solo dei due operandi: ad eseguire la conversione sull'altro provvederebbe automaticamente il compilatore)

La variabile promosso Arricchiamo il programma, come richiesto, introducendo una variabile `promosso`, cui verrà assegnato valore `vero` se la media è maggiore o eguale a 6, e valore `falso` in caso contrario.

```
#include <iostream>
int main(){
   int voto [3];
   int peso [3];
   double media;
   bool promosso;

   //Inserimento dati da parte dell'utente
   std::cout << "Inserisci il primo voto: ";
   std::cin >> voto[0];

   std::cout << "Inserisci il primo peso: ";
   std::cin >> peso[0];

   //Eccetera...

   //Calcoli
   media = ((double)(voto[0]*peso[0]+voto[1]*peso[1]+voto[2]*peso[2]))
           /(peso[0]+peso[1]+peso[2]);
   promosso = (media>=6);

   //Scrittura a terminale
   std::cout << "Media: ";
   std::cout << media;
   std::cout << "Promosso? ";
   std::cout << promosso;

   return 0;
}
```

Nel codice abbiamo fatto ricorso all'operatore binario di confronto `>=`, che accetta due operandi di uno stesso tipo numerico e restituisce `vero` (cioè 1) qualora il primo sia maggiore o eguale del secondo e `falso` (cioè 0) in caso contrario.

7 Gittata

Si scriva un programma che, a partire da angolo di alzo (in gradi sessagesimali) e velocità iniziale (in km/h) immessi dall'utente, restituisca, espressa in chilometri, la gittata di un cannone (nell'ipotesi che il moto del proiettile sia parabolico).

7.1 Soluzione

Nel moto parabolico, la gittata si calcola come:

$$\Delta x = \frac{v_0^2}{g} \sin 2\vartheta$$

dove v_0 è la velocità iniziale, g è l'accelerazione di gravità, e ϑ è l'angolo di alzo. Ricordiamo che in C++:

- per calcolare il quadrato di un numero, è possibile, alternativamente, moltiplicarlo per sé stesso, oppure ricorrere alla funzione potenza `pow(base,esponente)` messa a disposizione dal pacchetto `cmath`;
- per calcolare il seno di un angolo, è possibile utilizzare la funzione `sin(angolo)` messa a disposizione del pacchetto `cmath`, che calcola il seno di un angolo **fornito in radianti**.

Se l'utente immettesse la velocità in m/s e l'angolo in radianti, e se la gittata andasse restituita in metri, il programma potrebbe essere scritto semplicemente così:

```
#include <iostream>
#include <cmath>
int main(){
   double v0; //velocita' iniziale (in m/s)
   double g; //accelerazione (in m/s^2)
   double theta; //angolo di alzo (in radianti)
   double gittata; //gittata (in metri)

   //Imposto l'accelerazione di gravita'
   g = 9.81;

   //Chiedo all'utente v0 e theta
   std::cout << "Inserisci la velocita' iniziale (in m/s): "
   std::cin >> v0;
   std::cout << "Inserisci l'angolo di alzo (in radianti): "
   std::cin >> theta;

   //Calcolo la gittata (in metri)
   gittata = pow(v0,2)/g*sin(2*theta);

   //Scrivo il risultato a terminale
   std::cout << "La gittata vale: " << gittata << " m";

```

```
24    return 0;
25 }
```

Se vogliamo però adeguarci alle unità di misura richieste dall'esercizio (angolo e angolo di alzo forniti dall'utente rispettivamente in gradi e km/h, gittata da restituire in km), abbiamo davanti più possibilità:

- Conservare internamente le grandezze in questione (angolo, velocità iniziale e gittata) usando le unità di misura richieste, adeguando di conseguenza la formula:

```
1  #define _USE_MATH_DEFINES
2  #include <iostream>
3  #include <cmath>
4  int main(){
5     double v0; //velocita' iniziale (in km/h)
6     double g; //accelerazione (in m/s^2)
7     double theta; //angolo di alzo (in gradi)
8     double gittata; //gittata (in chilometri)
9
10    //Imposto l'accelerazione di gravita'
11    g = 9.81;
12
13    //Chiedo all'utente v0 e theta
14    std::cout << "Inserisci la velocita' iniziale (in km/h): "
15    std::cin >> v0;
16    std::cout << "Inserisci l'angolo di alzo (in gradi): "
17    std::cin >> theta;
18
19    //Calcolo la gittata (in chilometri!)
20    gittata = 0.001*pow(v0/3.6,2)/g*sin(2*theta*M_PI/180);
21
22    //Scrivo il risultato a terminale
23    std::cout << "La gittata vale: " << gittata << " km";
24
25    return 0;
26 }
```

- Mantenere internamente l'uso delle unità SI per le tre grandezze (cioè angolo in radianti, velocità iniziale in m/s e gittata in m), intervenendo con le opportune conversioni al momento dell'input e dell'output:

```
1  #define _USE_MATH_DEFINES
2  #include <iostream>
3  #include <cmath>
4  int main(){
5     double v0; //velocita' iniziale (in m/s)
6     double g; //accelerazione (in m/s^2)
7     double theta; //angolo di alzo (in radianti)
8     double gittata; //gittata (in metri)
9
```

```
10    //Imposto l'accelerazione di gravita'
11    g = 9.81;
12
13    //Chiedo all'utente v0 (l'utente digitera' il dato in km/h)
14    std::cout << "Inserisci la velocita' iniziale (in km/h): "
15    std::cin >> v0;
16
17    //Correggo subito il valore conservato in memoria per v0,
18    //cosi' da portarlo in m/s
19    v0 = v0/3.6;
20
21    //Chiedo all'utente theta (l'utente digitera' il dato in gradi)
22    std::cout << "Inserisci l'angolo di alzo (in gradi): "
23    std::cin >> v0;
24
25    //Correggo subito il valore conservato in memoria per theta,
26    //cosi' da portarlo in radianti
27    theta = theta*M_PI/180;
28
29    //Calcolo la gittata (in metri!)
30    gittata = pow(v0,2)/g*sin(2*theta);
31
32    //Scrivo il risultato a terminale (in chilometri!)
33    std::cout << "La gittata vale: " << gittata/1000 << " km";
34
35    return 0;
36 }
```

Si osservi che, nei codici presentati, per indicare π abbiamo adoperato la costante chiamata `M_PI` messa a nostra disposizione dalla libreria `cmath`. Per abilitare l'uso delle costanti matematiche nel programma (come `M_PI` per π ed `M_E` per e) occorre però premettere all'inclusione della libreria `cmath` la direttiva:

```
#define _USE_MATH_DEFINES
```

Scrivere direttamente 3.14 (o 3.14159) all'interno del codice, così come usare 9.81 direttamente nella formula per il calcolo della gittata, evitando l'introduzione della variabile `g`, sono scelte **sconsigliabili** (e la ragione è esattamente quella stessa che suggerisce di preferire le "lettere" ai valori numerici quando si scrive una qualunque formula matematica o fisica!).

8 Numero di Nepero

La costante matematica $e = 2,7182818284\ldots$ può essere approssimata per eccesso e per difetto nel modo seguente:

$$\left(1+\frac{1}{n}\right)^{n} < e < \left(1-\frac{1}{n}\right)^{-n}$$

e, al crescere di $n \in \mathbb{N}$, si ottengono approssimazioni sempre migliori. Scrivere un programma che accetti in ingresso n (intero 64 bit), calcoli le due approssimazioni (double) e le scriva a terminale (stampando il risultato in notazione esponenziale con 16 cifre decimali). Si provi ad eseguire il programma immettendo, come valore di n, dieci, cento, mille, un milione (10^6), un miliardo (10^9), un milione di miliardi (10^{12}), un miliardo di miliardi (10^{18}). Cosa si osserva? Perché?

8.1 Soluzione

```
1  #include <iostream>
2  #include <iomanip>
3  #include <cmath>
4  int main(){
5     long long n;
6     double approx_difetto, approx_eccesso;
7     std::cout << std::scientific;
8     std::cout << std::setprecision(16);
9
10    std::cout << "Inserisci n: ";
11    std::cin >> n;
12
13    approx_difetto = pow(1+1/(double)n,n);
14    approx_eccesso = pow(1-1/(double)n,-n);
15
16    std::cout << "Appross. per difetto di e: " << approx_difetto << "\n";
17    std::cout << "Appross. per eccesso di e: " << approx_eccesso << "\n";
18
19    return 0;
20 }
```

Si osservi l'uso dell'operatore `(double)` per eseguire una conversione esplicita di tipo `long long`→`double`, necessaria per forzare l'uso della divisione tra numeri `double` (in luogo della divisione euclidea tra numeri interi) alle righe 13 e 14. Si poteva ottenere lo stesso risultato, in modo più sintetico, utilizzando come numeratore della divisione, al posto di 1, il numero 1.0 (che viene interpretato dal compilatore come numero `double` e quindi costringe il computer ad usare la divisione di numeri `double`):

```
approx_difetto = pow(1+1.0/n,n);
approx_eccesso = pow(1-1.0/n,-n);
```

Di seguito riportiamo le approssimazioni restituite dal calcolatore per i valori di `n` proposti dal testo dell'esercizio:

n	Per difetto	(distanza da e)	Per eccesso	(distanza da e)
10	2.5937424601000023	-5%	2.8679719907924408	$+6\%$
100	2.7048138294215285	-0.5%	2.7319990264290284	$+0.6\%$
1000	2.7169239322355936	-0.05%	2.7196422164428529	$+0.05\%$
10^6	2.7182804690957534	-0.00005%	2.7182831876793716	$+0.00005\%$
10^9	2.7182820520115603	$+0.000008\%$	2.7182817529399266	-0.00003%
10^{12}	2.7185234960372378	$+0.009\%$	2.7182216960557022	-0.002%
10^{18}	1.0000000000000000	-60%	1.0000000000000000	-60%

La teoria ci assicura che:

Teorema. *La successione $a_n = (1 + \frac{1}{n})^n$ è monotona strettamente crescente; la successione $b_n = (1 - \frac{1}{n})^{-n}$ è monotona strettamente decrescente; le due successioni forniscono, al crescere di n, approssimazioni sempre migliori (rispettivamente, per difetto e per eccesso) di un medesimo numero reale, detto numero di Nepero ed indicato con e.*

Eseguendo il programma, osserviamo invece che:

- fintanto che n si mantenga abbastanza piccolo, il comportamento osservato è proprio quello che ci attendiamo: $\left(1 + \frac{1}{n}\right)^n$ cresce rimanendo minore di e e avvicinandosi sempre di più ad esso; $\left(1 - \frac{1}{n}\right)^{-n}$ decresce rimanendo maggiore di e e avvicinandosi sempre di più ad esso;

- già per $n = 10^9$ osserviamo un comportamento anomalo: il valore calcolato dal computer per $\left(1 + \frac{1}{n}\right)^n$ supera e, mentre il valore calcolato per $\left(1 - \frac{1}{n}\right)^{-n}$ è inferiore ad e;

- se lasciamo crescere n ancora, assistiamo ad un degradarsi della precisione delle due approssimazioni (che, stando alla teoria, dovrebbero invece migliorare sempre più!), con esiti apertamente catastrofici nel caso $n = 10^{18}$.

La spiegazione di questo comportamento anomalo del programma va ancora ricercata negli errori di arrotondamento che gravano ogni calcolo il computer esegua con i numeri reali.

Analizziamo per esempio cosa accade nel calcolo di $a_n = (1+\frac{1}{n})^n$: il computer deve anzitutto calcolare la base $b = 1+\frac{1}{n}$: nel compiere questa operazione, otterrà un risultato *leggermente sbagliato*, che indicheremo con $\widetilde{b} = b + \mathrm{err}_1$. Questo $\widetilde{b}$ così ottenuto verrà poi elevato dal computer alla n-esima potenza; anche questa operazione porterà con sé un certo errore di arrotondamento err_2, cosicché il computer otterrà, come valore di a_n, il numero $\widetilde{a_n} = \left(\widetilde{b}\right)^n + \mathrm{err}_2 = (b+\mathrm{err}_1)^n + \mathrm{err}_2 = \left(1 + \frac{1}{n} + \mathrm{err}_1\right)^n + \mathrm{err}_2$.

Orbene: immaginiamo pure che gli errori che abbiamo denotato con err_1 ed err_2 siano entrambi molto piccoli; questo non impedisce che $\widetilde{a_n}$ possa differire anche moltissimo da a_n. Per intuirne la ragione, si rammenti che elevare alla n-esima potenza un numero significa moltiplicarlo per sé stesso n volte; l'errore err_1, se pur molto piccolo, è quindi presente in ciascuno degli n fattori che stiamo moltiplicando; per grandi valori di n, può quindi alterare significativamente il valore della potenza.

In generale, se anche l'errore introdotto da ogni calcolo è piccolo, la *propagazione* degli errori può comunque degradare gravemente la precisione del risultato.

9 Esercizi aggiuntivi

9.1 Normalizzare angoli

All'inizio del corso, abbiamo discusso della divisione euclidea tra due numeri interi: ricordiamo che, dati $a, b \in \mathbb{Z}$, con $b \neq 0$, il risultato della divisione euclidea è quell'unica coppia di numeri $q \in \mathbb{Z}, r \in \{0, \ldots, b-1\}$ tali che $a = bq + r$.

Si può similmente anche introdurre una *divisione euclidea di numeri reali*, nel modo seguente: dati $a, b \in \mathbb{R}$, con $b \neq 0$, si dice risultato della divisione euclidea reale quell'unica coppia di numeri $q \in \mathbb{Z}, r \in [0, b)$ tali che $a = bq + r$. Per esempio:

- nella divisione euclidea reale per 1, il quoziente è la parte intera $\lfloor a \rfloor \in \mathbb{Z}$ del numero, mentre il resto è la sua parte frazionaria $\{a\} \in [0, 1)$;
- eseguire la divisione euclidea reale di un angolo per 2π significa esprimerlo come un certo numero intero di angoli giri (quoziente), più un angolo dell'intervallo $[0, 2\pi)$ (resto): per esempio, $27/4\pi =$ tre angoli giri $+ 3\pi/4$

Per calcolare il resto di una divisione euclidea tra numeri reali, il C++ mette a nostra disposizione (nella libreria `cmath`) la funzione

```
fmod(dividendo, divisore)
```

che accetta due argomenti `double` e restituisce un valore `double`.

Si scriva un programma che, leggendo da terminale un angolo in gradi (non necessariamente intero) immesso dall'utente, ne calcoli:

- l'angolo equivalente nell'intervallo $[0°, 360°)$
- l'angolo equivalente nell'intervallo $[-180°, 180°)$

Si scriva poi un programma che assolva ad un compito analogo, però in radianti.

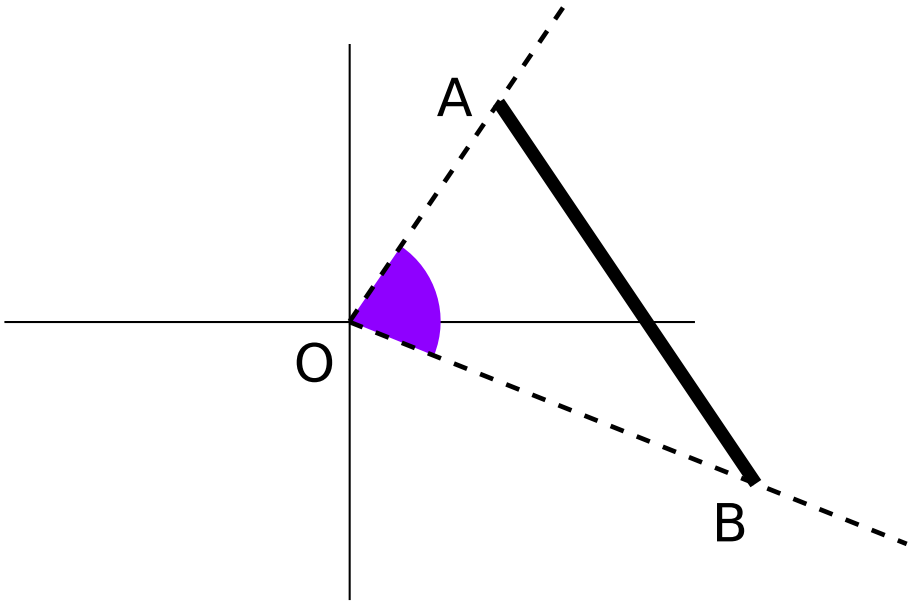
9.2 Segmenti visti da punti

Dato un punto $P = (x, y)$ sul piano cartesiano, la funzione del pacchetto `cmath`

```
atan2(y, x)
```

restituisce l'angolo dell'intervallo $[-\pi, \pi)$ compreso tra il semiasse positivo delle ascisse e $\overrightarrow{OP}$ (sia i due argomenti accettati dalla funzione che il risultato restituito sono numeri `double`).

Si scriva un programma che, chiesti all'utente gli estremi A e B di un segmento $\overrightarrow{AB}$ **orientato** e non passante per l'origine, restituisca l'angolo in radianti (compreso tra $-\pi$ e π) sotto il quale l'origine vede $\overrightarrow{AB}$:



L'angolo sotto cui l'origine vede $\overrightarrow{AB}$ è quello tra $\overrightarrow{OA}$ e $\overrightarrow{OB}$; nella figura, per esempio, misura $-77°$.

Si arricchisca il programma in modo che il risultato ottenuto venga scritto a terminale anche in gradi, e si provi ad eseguirlo immettendo gli estremi dei segmenti seguenti:

- $(0, 1) \to (1, 0)$ (dovrebbe restituire $-90°$)
- $(1, 0) \to (0, 1)$ (dovrebbe restituire $+90°$)
- $(1, 1) \to (-1, 0)$ (dovrebbe restituire $+135°$)
- $(-1, -1) \to (-1, 1)$ (dovrebbe restituire $-90°$)

9.3 Dentro e fuori

Riutilizzando il codice prodotto svolgendo l'esercizio precedente, si scriva un programma che, dati i vertici A, B e C di un triangolo (i cui lati non passino per l'origine), di cui l'utente immette a terminale le coordinate:

- calcoli gli angoli (compresi tra $-\pi$ e π) sotto i quali l'origine vede i segmenti orientati $\overrightarrow{AB}$, $\overrightarrow{BC}$ e $\overrightarrow{CA}$;
- sommi i tre angoli così ottenuti;
- scriva tale somma a terminale (in radianti e in gradi).

Quanto ti aspetti faccia la somma, nel caso in cui l'origine appartenga al triangolo? E nel caso in cui l'origine non vi appartenga? Esegui il programma immettendo opportuni dati di test per verificare la tua ipotesi.

4 Strutture di controllo

1 Parole magiche

Il seguente programma, dopo aver chiesto all'utente di immettere due numeri interi, restituisce una parola magica (non necessariamente dotata di significato!) composta usando le lettere A,B,...,L. Rispondere alle domande seguenti **senza trascrivere il programma a computer** (le domande contrassegnate con * sono opzionali).

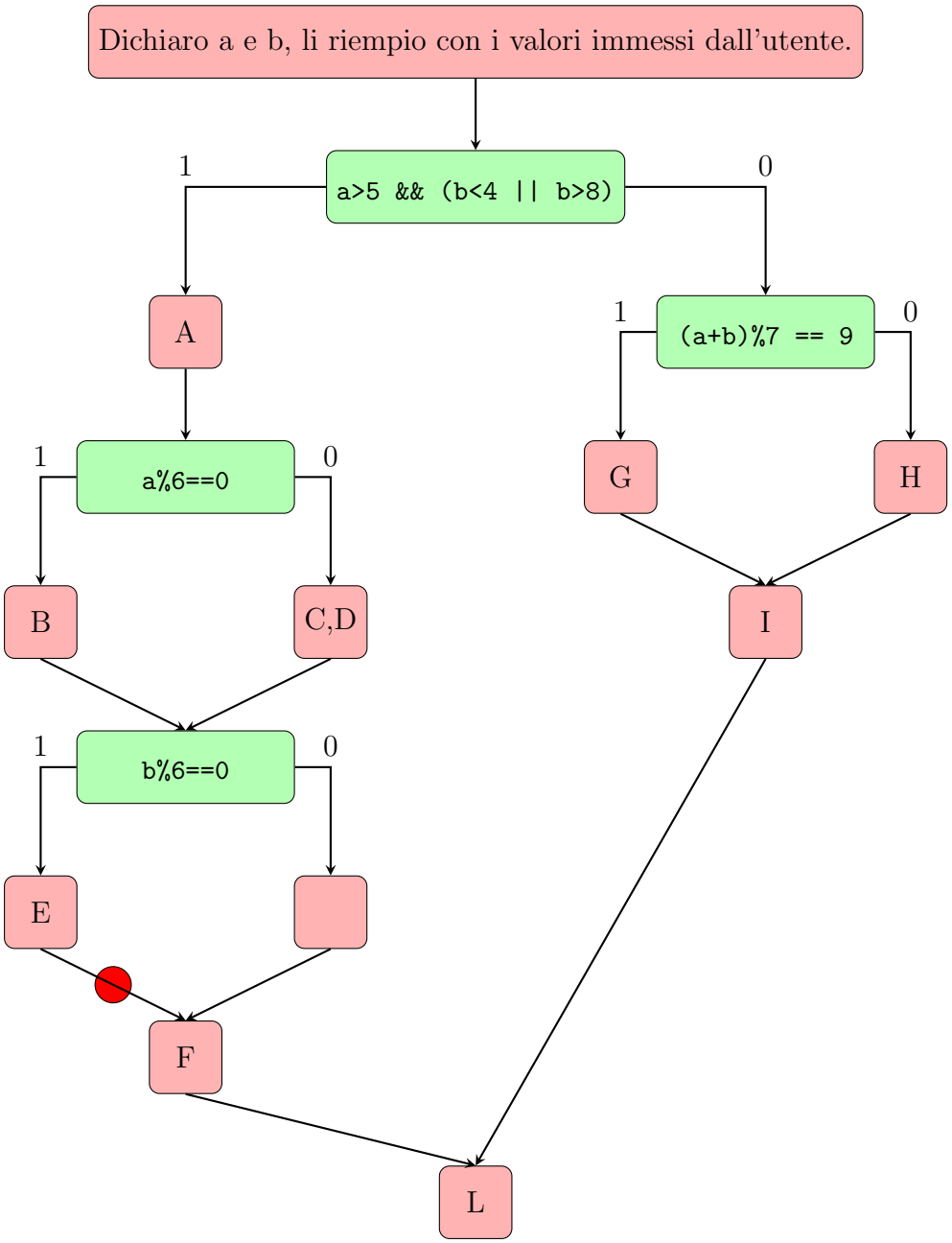
```
#include <iostream>
int main(){
   int a, b;
   std::cout << "Dammi due interi a ed b: ";
   std::cin >> a >> b;
   std::cout << "Parola magica: ";

   if (a>5 && (b<4 || b>8)){
      std::cout << "A";
      if(a%6==0){
         std::cout << "B";
      }else{
         std::cout << "C";
         std::cout << "D";
      }
      if(b%6==0){
         std::cout << "E";
         return 0;
      }
      std::cout << "F";
   }else{
      if((a*b)%7==9){
         std::cout << "G";
      }else{
         std::cout << "H";
      }
      std::cout << "I";
   }
   std::cout << "L";
   return 0;
}
```

1. Quale parola viene mostrata a video se l'utente immette i seguenti dati come valori di `a` e di `b`?
   a) 5 e 40
   b) 6 e 4
   c) 8 e 24
   d) 9 e 25
   e) 19 e 1
   f) 18 e 1
2. Può capitare che le seguenti coppie di lettere compaiano in una stessa parola magica? Perché?
   a) B e I
   b) B e C
   c) B ed E
   d) C ed E
   e) E ed F
3. Una delle dieci lettere non compare mai. Quale? Perché?
4. È vero che, se la E compare, allora è l'ultima lettera?
5. (*) È vero che le parole magiche non contenenti la A contengono la L? E il viceversa?
6. (*) È vero che ogni parola non contenente la E contiene la L? E il viceversa?
7. (*) È vero che ogni parola contenente la A contiene la E o la F?
8. (*) È vero che ogni parola magica ha sempre almeno due lettere?

Soluzione

Il programma può essere schematizzato con il seguente diagramma di flusso, nel quale:

- i riquadri verdi racchiudono le condizioni, vale a dire le espressioni booleane che il computer calcola per decidere quale ramo della selezione eseguire;
- il pallino rosso indica l'arresto del programma.



Guardando al diagramma, non è difficile rispondere alle domande proposte dal testo.

1. Quale parola viene mostrata a video se l'utente immette i seguenti dati come valori di `a` e di `b`?
   a) 5 e 40: HIL
   b) 6 e 4: HIL
   c) 8 e 24: ACDE
   d) 9 e 25: ACDFL
   e) 19 e 1: ACDFL
   f) 18 e 1: ABFL

2. Può capitare che le seguenti coppie di lettere compaiano in una stessa parola magica? Perché?
   a) B e I. Non può succedere, in quanto i comandi che scrivono tali lettere si trovano su rami opposti della selezione "esterna".
   b) B e C. Non può succedere, in quanto i comandi che scrivono tali lettere si trovano su rami opposti di una delle selezioni "interne".
   c) B ed E. Può capitare. Per esempio, si immagini che l'utente immetta 18 come valore di `a` e 24 come valore di `b`; il programma procede così:
      - calcola la condizione `a>5 && (b<4||b>8)`; ottiene come risultato VERO, e imbocca quindi il ramo vero della selezione esterna;
      - scrive A;
      - calcola la condizione `a%6==0`; ottiene come risultato VERO, e imbocca quindi il ramo vero di questa selezione interna: dunque scrive B, dopodiché tale ramo esaurisce e quindi la selezione interna in questione si conclude;
      - calcola la condizione `b%6==0`; ottiene come risultato VERO, e imbocca quindi il ramo vero di questa seconda selezione interna: dunque scrive E ed esegue `return 0`, che arresta l'esecuzione del programma.
   d) C ed E. Può capitare (vedi punto 1 dell'esercizio)
   e) E ed F. Non può succedere; l'istruzione che scrive E è infatti immediatamente seguita da un `return 0`, che arresta l'esecuzione del programma e impedisce che esso arrivi a scrivere F.

3. Una delle dieci lettere non compare mai. Quale? Perché?

   La lettera G non viene mai scritta, in quanto la condizione `(a+b)%7==9` dà, come risultato, sempre falso (il resto di un qualunque intero nella divisione per 7 è un numero compreso tra 0 e 6: non può in alcun caso risultare uguale a 9).

4. È vero che, se la E compare, allora è l'ultima lettera?

   Sì, perché la scrittura di E è immediatamente seguita da un `return 0`, che arresta l'esecuzione del programma.

5. (*) È vero che le parole magiche non contenenti la A contengono la L? E il viceversa?

- Se il programma non scrive A, vuol dire che non ha percorso il ramo vero della selezione esterna, cioè che il calcolo della condizione `a>5 && (b<4 || b>8)` dà come risultato `FALSO`. Il ramo della selezione esterna ad essere percorso è dunque quello falso; conclusasi la sua esecuzione, la selezione esterna si chiude e immancabilmente viene scritta L. Quindi: se la parola magica non contiene A, sicuramente contiene L.
- Viceversa, si supponga che la parola magica contenga L: questo non dice nulla sul fatto che il ramo percorso dal programma, nell'ambito della selezione più esterna, sia quello vero o quello falso; così, A potrebbe essere stata scritta, ma potrebbe anche non essere stata scritta. Quindi: se la parola magica contiene L, potrebbe contenere come non contenere la A (si vedano anche gli esempi di cui al punto 1 di questo esercizio).

6. (*) È vero che ogni parola non contenente la E contiene la L? E il viceversa?
   L'esecuzione di un programma termina sempre con un `return`; in questo programma, i punti di uscita possibili sono due, e sono immediatamente preceduti, rispettivamente, dalla scrittura di E e dalla scrittura di L. Quindi la parola magica prodotta contiene sempre una, ed una soltanto, di queste due lettere.

7. (*) È vero che ogni parola contenente la A contiene la E o la F?
   Sì, è vero. La presenza di A segnala che il programma ha scelto il ramo vero della selezione esterna. Si trova allora prima ad affrontare la sotto-selezione `a%6==0`; conclusasi questa, si imbatte nella seconda sotto-selezione `b%6==0`...
   - se quest'ultima condizione dà risultato vero, il programma scrive E e termina;
   - in caso contrario, la sotto-selezione si conclude (il suo ramo falso è infatti vuoto); il programma si ritrova dunque a scrivere F, esaurendo così il ramo vero della selezione esterna; scrive quindi L e termina.

   Dunque ogni parola magica contenente A contiene una, ed una soltanto, delle due lettere E ed L.

8. (*) È vero che ogni parola magica ha sempre almeno due lettere?
   Guardando ai percorsi di esecuzione possibili, è facile accorgersi che ogni parola magica ha sempre almeno tre lettere.

2 Equazioni di secondo grado

Un'equazione della forma $ax^2 + bx + c = 0$ ha un numero variabile di soluzioni (sul campo reale), a seconda dei valori dei coefficienti. In particolare, può capitare che:

- esistano due soluzioni distinte;
- esistano due soluzioni coincidenti;
- la soluzione esista e sia unica;
- l'equazione sia impossibile;
- ogni numero reale sia soluzione.

Si scriva un programma che, dati i tre coefficienti a, b e c (che dovranno essere tre numeri interi `int` digitati dall'utente), indichi (scrivendo a terminale un opportuno messaggio) quale dei cinque casi sopra elencati si verifica.

Soluzione

Dobbiamo anzitutto discutere il problema da un punto di vista matematico. È opportuno distinguere vari casi, a seconda del grado del polinomio:

- se il polinomio ha grado 2 (cioè $a \neq 0$), è possibile determinare il numero di radici guardando al discriminante $\Delta = b^2 - 4ac$:
  - se $\Delta > 0$, abbiamo due radici distinte;
  - se $\Delta = 0$, abbiamo due radici coincidenti;
  - se $\Delta < 0$, il polinomio è privo di radici.
- se il polinomio ha grado 1 (cioè $a = 0$, ma $b \neq 0$), la radice esiste unica;
- se il polinomio ha grado 0 (cioè $a = 0$ e $b = 0$, ma $c \neq 0$) non ammette alcuna radice.
- se il polinomio è il polinomio nullo (cioè $a = 0$, $b = 0$ e $c = 0$), ogni numero reale è radice.

Un modo possibile di implementare questa divisione in casi in C++ è riportato qui sotto. I commenti segnalano, nei diversi punti del programma, quali informazioni abbiamo a disposizione sul polinomio: la cascata di successioni annidate rende via via più precisa la conoscenza del polinomio, sinché non diventa possibile fornire all'utente una risposta sul numero di radici.

```cpp
#include <iostream>
int main(){
   int a, b, c; //Coefficienti di ax^2 + bx + c
   int delta; //Discriminante

   std::cout << "Immetti nell'ordine i tre coefficienti: ";
   std::cin >> a >> b >> c;

   if(a!=0){
      //Grado 2
      delta = b*b - 4*a*c;
      if(delta>=0){
         //Grado 2, discrim. positivo o nullo
         if(delta>0){
            //Grado 2, discrim. positivo
            std::cout << "Due radici distinte";
         }else{
            //Grado 2, discrim. nullo
            std::cout << "Due radici coincidenti";
         }
      }else{
         //Grado 2, discrim. negativo
         std::cout << "Nessuna radice";
      }
   }else{
      //Grado 0, 1, o pol. nullo
      if(b!=0){
         //Grado 1
         std::cout << "Una e una sola radice";
      }else{
         //Grado 0, o pol. nullo
         if(c!=0){
            //Grado 0
            std::cout << "Nessuna radice";
         }else{
            //Pol. nullo
            std::cout << "Ogni numero reale e' radice";
         }
      }
   }
   return 0;
}
```

La casistica che vogliamo implementare relativamente al discriminante è ternaria (i casi sono $\Delta > 0$, $\Delta = 0$, $\Delta < 0$); tuttavia, mancando in C++ una selezione a tre vie, è necessario simularla usando due selezioni annidate: la scelta sopra proposta (distinguere anzitutto il caso $\Delta \geq 0$ dal caso $\Delta < 0$, e poi dividere il primo nei due sotto-casi $\Delta > 0$ e $\Delta = 0$) è solo una delle molte possibilità equivalenti.

3 Due rette nel piano

Si scriva un programma che, date le equazioni di due rette nel piano

$$r_1 : \ a_1x + b_1y + c_1 = 0 \qquad r_2 : \ a_2x + b_2y + c_2 = 0$$

che l'utente fornisce digitando a terminale i sei coefficienti (numeri `int`),

1. verifichi che r_1 ed r_2 siano effettivamente due rette e, in caso contrario, arresti l'esecuzione scrivendo un messaggio d'errore a terminale;
2. comunichi all'utente se le due rette sono coincidenti, parallele e distinte, oppure incidenti.

Soluzione

Anzitutto, alcune premesse da un punto di vista matematico.

Definizione. *Date due n-uple (non nulle) di numeri* $(\alpha_1, \ldots, \alpha_n)$ *e* $(\beta_1, \ldots, \beta_n)$*, esse si dicono* proporzionali *se coincidono a meno di un riscalamento, cioè se esiste* $\lambda \in \mathbb{R} \setminus \{0\}$ *tale che:*

$$(\beta_1, \ldots, \beta_n) = \lambda(\alpha_1, \ldots, \alpha_n)$$

cioè:

$$\beta_1 = \lambda\alpha_1, \quad \beta_2 = \lambda\alpha_2, \quad \ldots, \quad \beta_n = \lambda\alpha_n$$

Per esempio:

- $(5, 10, 20)$ e $(3, 6, 12)$ sono proporzionali: moltiplicando la prima per $\lambda = 3/5$ si ottiene la seconda;
- $(5, 0, 1)$ e $(10, 0, 2)$ sono proporzionali: moltiplicando la prima per $\lambda = 2$ si ottiene la seconda;
- $(1, 2)$ e $(1, 3)$ non sono proporzionali: non esiste nessun $\lambda \in \mathbb{R} \setminus \{0\}$ per cui si abbia $\lambda \cdot 1 = 1$ e contemporaneamente $\lambda \cdot 2 = 3$.

Il parallelismo e la coincidenza di due rette nel piano possono essere studiati guardando all'esistenza di certe relazioni di proporzionalità tra i loro coefficienti.

Teorema (Caratterizzazione della coincidenza e del parallelismo di rette nel piano)**.** *Date due rette nel piano, di equazioni*

$$r_1 : \ a_1x + b_1y + c_1 = 0 \qquad r_2 : \ a_2x + b_2y + c_2 = 0$$

si ha che:

- r_1 *e* r_2 *sono parallele* $\iff$ *le due coppie* (a_1, b_1) *e* (a_2, b_2) *sono proporzionali;*
- r_1 *e* r_2 *coincidono* $\iff$ *le due terne* (a_1, b_1, c_1) *e* (a_2, b_2, c_2) *sono proporzionali.*

Il programma da scrivere può essere quindi disegnato secondo uno schema di questo tipo:

- chiedo all'utente di immettere i sei coefficienti e li memorizzo dentro opportune variabili;
- controllo se r_1 è effettivamente una retta: se non lo è, arresto l'esecuzione del programma;
- controllo se r_2 è effettivamente una retta: se non lo è, arresto l'esecuzione del programma;
- guardo se (a_1, b_1) e (a_2, b_2) sono proporzionali:
  - se lo sono, allora mi trovo davanti a due rette parallele; se voglio distinguere il caso "coincidenti" da quello "parallele e distinte", devo guardare se (a_1, b_1, c_1) e (a_2, b_2, c_2) sono proporzionali:
    - se lo sono, allora le due rette sono parallele e coincidenti;
    - in caso contrario, le due rette sono parallele e distinte.
  - in caso contrario, le due rette non sono parallele, cioè sono incidenti.

Rimane ancora da discutere una sola questione, cioè come determinare, operativamente, se due n-uple (non nulle) di numeri reali date $(\alpha_1, \ldots, \alpha_n)$ e $(\beta_1, \ldots, \beta_n)$ siano proporzionali o meno. L'intuito suggerisce che basti controllare l'uguaglianza dei rapporti tra elementi corrispondenti:

$$\frac{\alpha_1}{\beta_1} = \frac{\alpha_2}{\beta_2} = \cdots = \frac{\alpha_n}{\beta_n}$$

Un simile approccio presenta però dei seri limiti:

- da un punto di vista matematico, non è in generale assicurato che i rapporti si possano calcolare: i denominatori potrebbero annullarsi;
- da un punto di vista della programmazione, la divisione costringe, anche nel caso in cui tutti i numeri α_i e β_i siano interi, ad impiegare l'aritmetica `double`; in particolare, il criterio richiede di fatto di valutare l'eguaglianza di n numeri di tipo `double`, e i pericoli che una simile operazione comporta dovrebbero ormai essere ben noti (si veda l'esercizio 05 del capitolo 3!)

Per valutare la proporzionalità di due n-uple di numeri, un criterio meno intuitivo, ma più pulito ed efficace, è questo:

Teorema. *Due n-uple non nulle di numeri reali $(\alpha_1, \ldots, \alpha_n)$ e $(\beta_1, \ldots, \beta_n)$ sono proporzionali se, e solamente se, per ogni scelta di indici distinti i e j, i due prodotti incrociati $\alpha_i\beta_j$ e $\alpha_j\beta_i$ coincidono. In particolare:*

- *(α_1, α_2) e (β_1, β_2) proporzionali $\iff \alpha_1\beta_2 = \alpha_2\beta_1$;*
- *$(\alpha_1, \alpha_2, \alpha_3)$ e $(\beta_1, \beta_2, \beta_3)$ proporzionali $\iff \alpha_1\beta_2 = \alpha_2\beta_1$, $\alpha_1\beta_3 = \alpha_3\beta_1$ e $\alpha_2\beta_3 = \alpha_3\beta_2$;*
- *e così via...*

Il nostro programma, in conclusione, potrà essere scritto così:

```
#include <iostream>
int main(){
   int a1, b1, c1; //Coefficienti retta r1 (a1 x + b1 y + c1 = 0)
   int a2, b2, c2; //Coefficienti retta r2 (a2 x + b2 y + c2 = 0)

   //Inserimento dati da tastiera
   std::cout << "Inserisci i coefficienti della prima retta: ";
   std::cin >> a1 >> b1 >> c1;
   std::cout << "Inserisci i coefficienti della seconda retta: ";
   std::cin >> a2 >> b2 >> c2;

   //Trappola: mi assicuro che r1 sia davvero una retta
   if(a1==0 && b1==0){
      std::cout << "r1 non e' una retta";
      return 0;
   }

   //Trappola: mi assicuro che r2 sia davvero una retta
   if(a2==0 && b2==0){
      std::cout << "r2 non e' una retta";
      return 0;
   }

   //Se arrivo qui sono sicuro che r1 e r2 siano rette

   if(a1*b2==b1*a2){
      //Le due rette sono parallele
      if(a1*c2==c1*a2 && b1*c2==c1*b2){
         //Le due rette coincidono
         std::cout << "Rette coincidenti";
      }else{
         //Le due rette sono parallele e distinte
         std::cout << "Rette parallele e distinte";
      }
   }else{
      //Le due rette sono incidenti
      std::cout << "Rette incidenti";
   }

   return 0;
}
```

4 Cicli

Si studi il modo in cui vengono eseguiti i seguenti programmi, e si dica quale output viene prodotto a terminale da ciascuno di essi... Inoltre, uno dei quattro programmi contiene un grave errore: quale?

```cpp
#include <iostream>
int main(){
   int i;
   i = 0;
   while(i<6){
      std::cout << i << " ";
      i = i+2;
   }
   std::cout << "\nCiao " << i;
   return 0;
}
```

```cpp
#include <iostream>
int main(){
   int i, somma;
   i = 0;
   while(i<=3){
      std::cout << i << " ";
      somma = somma+i;
      i = i+1;
   }
   std::cout << "Somma:" << somma;
   return 0;
}
```

```cpp
#include <iostream>
int main(){
   int i;
   i = 0;
   while(i>0 && i<20){
      std::cout << i << "\n";
      i = i+1;
   }
   std::cout << "Ciao " << i;
   return 0;
}
```

```cpp
#include <iostream>
int main(){
   int a, b;
   a = 0; b = 8;
   while(a<35 && b>3){
      std::cout << a << " ";
      std::cout << b << "\n";
      a = a+b;
      if(a%2==0){
         b = b-1;
      }
   }
   return 0;
}
```

Soluzione

Primo programma (in alto a SX) Il programma introduce una variabile `i`, il cui valore viene inizialmente impostato a 0. Seguiamo ora l'esecuzione del ciclo:

1. Il computer controlla se `i<6`: è vero! (`i`, infatti, vale 0). Allora esegue tutte le istruzioni del ciclo: scrive il valore di `i` (cioè 0) a terminale, poi modifica il valore di `i` incrementandolo di due unità: il valore di `i` è così diventato 2. Avendo esaurito le istruzioni del ciclo, il computer torna alla condizione di controllo...

2. Il computer controlla se `i<6`: è vero! (`i`, infatti, vale 2). Allora esegue tutte le istruzioni del ciclo: scrive il valore di `i` (cioè 2) a terminale, poi modifica il valore di `i` incrementandolo di due unità: il valore di `i` è così diventato 4. Avendo esaurito le istruzioni del ciclo, il computer torna alla condizione di controllo...

3. Il computer controlla se `i<6`: è vero! (`i`, infatti, vale 4). Allora esegue tutte le istruzioni del ciclo: scrive il valore di `i` (cioè 4) a terminale, poi modifica il valore di `i` incrementandolo di due unità: il valore di `i` è così diventato 6. Avendo esaurito le istruzioni del ciclo, il computer torna alla condizione di controllo...

4. Il computer controlla se `i<6`: è falso, in quanto `i` vale 6!

Vengono quindi eseguite in tutto 3 iterazioni del ciclo, e sul terminale vengono scritti, nell'ordine, i numeri 0, 2, e 4. Conclusasi l'esecuzione del ciclo, il computer procede oltre lungo l'esecuzione del programma: viene eseguita l'istruzione alla riga 9 che scrive a terminale la parola "Ciao" seguita dal valore di `i` (cioè 6); dopodiché il programma termina. Alla fine, a terminale possiamo leggere:

```
0 2 4
Ciao 6
```

Secondo programma (in basso a SX) Il programma dichiara anzitutto una variabile `i`, cui viene assegnato il valore 0. L'esecuzione raggiunge dunque la riga 5, che introduce il ciclo: si calcola quindi il valore di verità della condizione `i>0 && i<20`, e si ottiene `FALSO`. Le istruzioni del ciclo non vengono quindi eseguite nemmeno una volta: il programma passa oltre, raggiungendo la riga 9, scrive a terminale la parola "Ciao" seguita dal valore di `i` (cioè 0), quindi termina. Alla fine, sul terminale troveremo scritto soltanto:

```
Ciao 0
```

Terzo programma (in alto a DX) Il programma dichiara due variabili, `i` e `somma`; quindi, assegna ad `i` il valore 0. A questo punto, il computer si imbatte nel ciclo...

Il computer controlla anzitutto se sia vero o meno che `i<3`: è vero (`i`, infatti, vale 0)! Vengono allora eseguite, nell'ordine, tutte le istruzioni del ciclo: per prima cosa, il programma scrive il valore di `i` (cioè 0) sul terminale. Poi, si trova ad eseguire la riga 7, che chiede di leggere il valore della variabile `somma`, sommarci il valore di `i`, e impostare il risultato così ottenuto come nuovo valore di `somma`... Ecco, per l'appunto, il grave errore: il programma tenta di leggere il valore di `somma`, quando ancora `somma` non ha valore alcuno, ed è solo una "scatola vuota".

Non bisogna mai accedere in lettura una variabile *non ancora inizializzata*, cioè ancora priva di valore!

Per correggere il programma, possiamo aggiungere, prima dell'inizio del ciclo, un'istruzione di assegnazione `somma = 0`. Così corretto, il programma scrive a terminale i primi quattro numeri naturali (cioè 0, 1, 2 e 3) seguiti dalla loro somma (cioè 6):

```
0 1 2 3 Somma:6
```

Quarto programma (in basso a DX) Il quarto programma dichiara anzitutto due variabili `a` e `b`, e imposta, rispettivamente, 0 e 8 come loro valori. Giunto alla riga 6, si imbatte in un ciclo...

1. Il computer controlla se `a<35 && b>3`: è vero (infatti, `a` vale 0 e `b` vale 8, dunque si ottiene `VERO && VERO`, cioè `VERO`)! Vengono allora eseguite, una dopo l'altra, le istruzioni del ciclo:
   - Il programma scrive a terminale il valore di `a`, cioè 0;
   - Il programma scrive a terminale il valore di `b`, cioè 8;
   - Il programma calcola l'espressione `a+b`, ottenendo 8: tale risultato viene impostato come nuovo valore di `a`
   - Il programma si imbatte nella selezione della riga 10: la condizione `a%2==0` restituisce `VERO` (`a` infatti vale 8, e 8 è pari)! Viene allora percorso il ramo vero della selezione, cosicché il valore della variabile `b` viene diminuito di un'unità, e diventa 7

   Al termine dell'esecuzione di questa prima iterazione ciclo, `a` vale 8 e `b` vale 7. Avendo eseguito tutte le istruzioni del ciclo, il computer torna a valutare la condizione di controllo della riga 6...

2. Il computer controlla se `a<35 && b>3`: è vero (infatti, `a` vale 8 e `b` vale 7, dunque si ottiene `VERO && VERO`, cioè `VERO`)! Vengono allora eseguite, una dopo l'altra, le istruzioni del ciclo:
   - Il programma scrive a terminale il valore di `a`, cioè 8;
   - Il programma scrive a terminale il valore di `b`, cioè 7;
   - Il programma calcola l'espressione `a+b`, ottenendo 15: tale risultato viene impostato come nuovo valore di `a`
   - Il programma si imbatte nella selezione della riga 10: la condizione `a%2==0` restituisce `FALSO` (`a` infatti vale 15, e 15 è dispari)! Viene allora percorso il ramo falso della selezione (che però è vuoto, cioè privo di istruzioni).

   Al termine dell'esecuzione di questa seconda iterazione ciclo, `a` vale 15 e `b` vale 7. Avendo eseguito tutte le istruzioni del ciclo, il computer torna a valutare la condizione di controllo della riga 6...

3. Il computer controlla se `a<35 && b>3`: è vero (infatti, `a` vale 15 e `b` vale 7, dunque si ottiene `VERO && VERO`, cioè `VERO`)! Vengono allora eseguite, una dopo l'altra, le istruzioni del ciclo:
   - Il programma scrive a terminale il valore di `a`, cioè 15;
   - Il programma scrive a terminale il valore di `b`, cioè 7;
   - Il programma calcola l'espressione `a+b`, ottenendo 22: tale risultato viene impostato come nuovo valore di `a`

- Il programma si imbatte nella selezione della riga 10: la condizione `a%2==0` restituisce `VERO` (`a` infatti vale 22, e 22 è pari)! Viene allora percorso il ramo vero della selezione, cosicché il valore della variabile `b` viene diminuito di un'unità, e diventa 6.

Al termine dell'esecuzione di questa terza iterazione ciclo, `a` vale 22 e `b` vale 6. Avendo eseguito tutte le istruzioni del ciclo, il computer torna a valutare la condizione di controllo della riga 6...

4. Il computer controlla se `a<35 && b>3`: è vero (infatti, `a` vale 22 e `b` vale 6, dunque si ottiene `VERO && VERO`, cioè `VERO`)! Vengono allora eseguite, una dopo l'altra, le istruzioni del ciclo:
   - Il programma scrive a terminale il valore di `a`, cioè 22;
   - Il programma scrive a terminale il valore di `b`, cioè 6;
   - Il programma calcola l'espressione `a+b`, ottenendo 28: tale risultato viene impostato come nuovo valore di `a`
   - Il programma si imbatte nella selezione della riga 10: la condizione `a%2==0` restituisce `VERO` (`a` infatti vale 28, e 28 è pari)! Viene allora percorso il ramo vero della selezione, cosicché il valore della variabile `b` viene diminuito di un'unità, e diventa 5.

   Al termine dell'esecuzione di questa quarta iterazione ciclo, `a` vale 28 e `b` vale 5. Avendo eseguito tutte le istruzioni del ciclo, il computer torna a valutare la condizione di controllo della riga 6...

5. Il computer controlla se `a<35 && b>3`: è vero (infatti, `a` vale 28 e `b` vale 5, dunque si ottiene `VERO && VERO`, cioè `VERO`)! Vengono allora eseguite, una dopo l'altra, le istruzioni del ciclo:
   - Il programma scrive a terminale il valore di `a`, cioè 28;
   - Il programma scrive a terminale il valore di `b`, cioè 5;
   - Il programma calcola l'espressione `a+b`, ottenendo 33: tale risultato viene impostato come nuovo valore di `a`
   - Il programma si imbatte nella selezione della riga 10: la condizione `a%2==0` restituisce `FALSO` (`a` infatti vale 33, e 33 è dispari)! Viene allora percorso il ramo falso della selezione (che però è vuoto, cioè privo di istruzioni).

   Al termine dell'esecuzione di questa quinta iterazione ciclo, `a` vale 33 e `b` vale 5. Avendo eseguito tutte le istruzioni del ciclo, il computer torna a valutare la condizione di controllo della riga 6...

6. Il computer controlla se `a<35 && b>3`: è vero (infatti, `a` vale 33 e `b` vale 5, dunque si ottiene `VERO && VERO`, cioè `VERO`)! Vengono allora eseguite, una dopo l'altra, le istruzioni del ciclo:
   - Il programma scrive a terminale il valore di `a`, cioè 33;
   - Il programma scrive a terminale il valore di `b`, cioè 5;

- Il programma calcola l'espressione `a+b`, ottenendo 38: tale risultato viene impostato come nuovo valore di `a`
- Il programma si imbatte nella selezione della riga 10: la condizione `a%2==0` restituisce `VERO` (`a` infatti vale 38, e 38 è pari)! Viene allora percorso il ramo vero della selezione, cosicché il valore della variabile `b` viene diminuito di un'unità, e diventa 4.

Al termine dell'esecuzione di questa sesta iterazione ciclo, `a` vale 38 e `b` vale 4. Avendo eseguito tutte le istruzioni del ciclo, il computer torna a valutare la condizione di controllo della riga 6...

7. Il computer controlla se `a<35 && b>3`: è falso (infatti, `a` vale 38 e `b` vale 4, dunque si ottiene `FALSO && VERO`, cioè `FALSO`).

Vengono quindi eseguite in tutto sei iterazioni del ciclo. Completata l'esecuzione del ciclo, il computer passa oltre, raggiungendo la riga 14, ove il programma termina. Sul terminale troveremo alla fine scritto:

```
0 8
8 7
15 7
22 6
28 5
33 5
```

5 Divisbilità

Si scriva un programma che, dato un numero naturale $n \geq 2$ immesso dall'utente, ne mostri a terminale tutti i divisori naturali (si faccia uso del tipo numerico `int`). Si arricchisca in seguito il programma, in modo che:

- conti tutti i divisori del numero;
- scriva a terminale "e' primo" se il numero è primo; "e' composto" se il numero è composto;
- calcoli la somma di tutti i divisori del numero;
- indichi all'utente se il numero immesso è o meno *libero da quadrati* (un numero naturale n si dice *libero da quadrati* se non è diviso da nessun quadrato perfetto, all'infuori di 1).

Soluzione

Il punto di partenza è scrivere un programma che, scorrendo tutti gli interi tra 1 e n, verifichi, per ciascuno di essi, se si tratta di un divisore di n oppure no. Ancorché non esplicitamente richiesto dal testo, aggiungeremo preliminarmente anche una trappola per sincerarci che $n \geq 2$.

```
#include <iostream>
int main(){
   int n; //Il numero immesso dall'utente
   int candidato; //Il candidato divisore
   std::cout << "Inserisci il numero: ";
   std::cin >> n;

   //Trappola: controllo n sia un intero maggiore o uguale di 2
   if(n<2){
      std::cout << "Avevo chiesto un intero maggiore o uguale a 2!";
      return 0;
   }

   std::cout << "I divisori di " << n << " sono: ";
   candidato = 1;
   while(candidato<=n){
      if(n%candidato==0){
         //Il candidato e' davvero un divisore di n: lo scrivo a terminale
         std::cout << candidato << " ";
      }else{
         //Il candidato non e' un divisore di n, non devo fare nulla
      }
      //In vista della prox. iterazione, passo al candidato div. successivo
      candidato = candidato+1;
   }
   return 0;
}
```

Procediamo ora alle prime migliorie:

- per ottenere che il programma conti il numero di divisori, basterà introdurre una variabile `conto`, impostarne il valore inizialmente a 0 e incrementarla unitariamente per ogni divisore trovato nel corso dell'esecuzione del ciclo;
- per ottenere che il programma calcoli anche la somma di tutti i divisori, occorrerà un'ulteriore variabile `somma`, impostata inizialmente a zero e accresciuta successivamente di ogni divisore individuato;
- per capire se il numero è primo o composto, basterà controllare se `conto`, al termine del programma, è uguale a 2 o maggiore di 2.

Il codice diventa così:

```cpp
#include <iostream>
int main(){
   int n; //Il numero immesso dall'utente
   int candidato; //Il candidato divisore
   int somma; //Somma dei divisori
   int conto; //Numero dei divisori

   std::cout << "Inserisci il numero: ";
   std::cin >> n;

   //Trappola: controllo n sia un intero maggiore o uguale di 2
   if(n<2){
      std::cout << "Avevo chiesto un intero maggiore o uguale a 2!";
      return 0;
   }

   std::cout << "I divisori di " << n << " sono: ";

   candidato = 1; somma = 0; conto = 0;
   while(candidato<=n){
      if(n%candidato==0){
         //Il candidato e' davvero un divisore di n;
         //lo scrivo dunque a terminale
         std::cout << candidato << " ";

         //E poi aggiorno somma e conto
         somma = somma + candidato;
         conto = conto + 1;
      }else{
         //Il candidato non e' un divisore di n;
         //non devo fare nulla
      }

      //In vista della prossima iterazione, passo al
      //candidato divisore successivo
      candidato = candidato+1;
   }
   std::cout << "\n" << "Il numero di divisori e' " << conto << "\n";
   std::cout << "La somma dei divisori fa " << somma << "\n";
   if(conto==2){
      std::cout << "Il numero " << n << " e' primo";
   }else{
      std::cout << "Il numero " << n << " e' composto";
   }
   return 0;
}
```

Per esempio, se immettiamo 496 possiamo poi leggere a terminale:

```
Inserisci il numero: 496
I divisori di 496 sono: 1 2 4 8 16 31 62 124 248 496
Il numero di divisori e' 10
La somma dei divisori fa 992
Il numero 496 e' composto
```

Per ragioni di praticità, proponiamo un programma separato per risolvere l'ultimo punto dell'esercizio, vale a dire per decidere se n è libero o meno da quadrati: la nostra intenzione è di scorrere tutti i quadrati perfetti maggiori di 1 e minori di n...

```
#include <iostream>
int main(){
   int n; //Numero immesso dall'utente
   int i; //Variabile contatore: i*i e' il quadrato perfetto considerato
   bool quadrato_perfetto_divisore_trovato;

   std::cout << "Inserisci il numero: ";
   std::cin >> n;

   //Trappola: controllo n sia un intero maggiore o uguale di 2
   if(n<2){
      std::cout << "Avevo chiesto un intero maggiore o uguale a 2!";
      return 0;
   }

   i = 2;
   quadrato_perfetto_divisore_trovato = 0;
   while(i*i<=n){
      if(n%(i*i)==0){
         //Il quadrato perfetto i*i che sto considerando divide n
         quadrato_perfetto_divisore_trovato = 1;
      }else{
         //Il quadrato perfetto i*i che sto considerando non divide n
      }

      //Incremento il contatore i, cosi' che la prossima
      //iterazione del ciclo consideri il quadrato perfetto
      //i*i successivo...
      i = i+1;
   }
   if(quadrato_perfetto_divisore_trovato){
      std::cout << "Il numero " << n << " non e' libero da quadrati";
   }else{
      std::cout << "Il numero " << n << " e' libero da quadrati";
   }
   return 0;
}
```

Analizziamo il ruolo, nel programma, della variabile booleana `quadrato_perfetto_divisore_trovato`: inizialmente, viene impostata uguale a `FALSO`. Ogni iterazione del ciclo:

- se trova un quadrato perfetto che divide n, imposta `quadrato_perfetto_divisore_trovato` a `VERO`;
- in caso contrario, lascia la variabile `quadrato_perfetto_divisore_trovato` immutata.

Quindi:

- se al termine del programma `quadrato_perfetto_divisore_trovato` è vera, significa che il ciclo ha trovato, *almeno una volta*, un quadrato perfetto $\texttt{i}^2$ che fosse divisore di n;
- se al termine del programma `quadrato_perfetto_divisore_trovato` è falsa, significa che il ciclo non ha *mai* trovato, tra tutti i quadrati perfetti $\texttt{i}^2$ che ha considerato, un divisore di n.

Una variabile booleana come `quadrato_perfetto_divisore_trovato`, introdotta per segnalare se una certa eventualità, lungo le iterazioni di un ciclo, si è verificata almeno una volta o non si è mai verificata, viene designata talvolta con il nome di *flag* (in italiano: *bandierina*).

Un'ultima osservazione è questa: una volta trovato un quadrato perfetto che divida n, non ha chiaramente più senso che il ciclo continui ad essere eseguito. Possiamo quindi rendere il programma più efficiente modificando la condizione di controllo del ciclo in:

```
i*i<=n && !quadrato_perfetto_divisore_trovato
```

Un'alternativa all'uso di un *flag*, in questo caso, è rappresentata dall'uso di *trappole* (cioè selezioni ad una via che arrestino il programma nel ramo vero) all'interno del ciclo:

```cpp
#include <iostream>
int main(){
   int n; //Numero immesso dall'utente
   int i; //Variabile contatore: i*i e' il quadrato perfetto considerato
   std::cout << "Inserisci il numero: ";
   std::cin >> n;

   //Trappola: controllo n sia un intero maggiore o uguale di 2
   if(n<2){
      std::cout << "Avevo chiesto un intero maggiore di 2!";
      return 0;
   }

   i = 2;
   while(i*i<=n){
      //Trappola!
      if(n%(i*i)==0){
         //Il quadrato perfetto i*i che sto considerando divide n
         //Quindi certamente n non e' libero da quadrati!
         //Lo comunico all'utente e arresto il programma
         std::cout << "Il numero " << n << " non e' libero da quadrati";
         return 0;
      }

      //Incremento il contatore i, cosi' che la prossima
      //iterazione del ciclo consideri il quadrato perfetto
      //i*i successivo...
      i = i+1;
   }

   //Se il programma e' sopravvissuto arrivando fin qui,
   //vuol dire che non e' mai caduto in trappola,
   //cioe' che nessuno dei quadrati i*i considerato divide n...
   //posso allora dire con certezza che n e' libero da quadrati!

   std::cout << "Il numero " << n << " e' libero da quadrati";
   return 0;
}
```

6 Numeri primi

Si scriva un programma, che, dato un numero naturale $n \geq 2$ immesso dall'utente, determini se è primo o composto (informandone l'utente con un messaggio a terminale). A questo scopo, si riutilizzi il codice dell'esercizio precedente, adattandolo e curandone l'efficienza.

Soluzione

Proponiamo una soluzione che fa uso di un *flag* (cfr. ultimo punto dell'esercizio precedente).

```
#include <iostream>
int main(){
   int n; //Numero immesso dall'utente
   int candidato; //Candidato divisore
   bool divisore_trovato; //Flag

   //Immissione del numero da parte dell'utente
   std::cout << "Inserisci un numero maggiore o uguale a 2: ";
   std::cin >> n;
   //Aggiungere qui eventuale trappola per assicurarsi che n>=2...

   candidato = 2; divisore_trovato = 0;
   while(candidato<n){
      if(n%candidato==0){
         divisore_trovato = 1;
      }
      candidato = candidato+1;
   }
   if(divisore_trovato){
      std::cout << "Il numero " << n << " e' composto";
   }else{
      std::cout << "Il numero " << n << " e' primo";
   }
   return 0;
}
```

Per rendere più efficiente il programma, possiamo ritoccare in senso restrittivo la condizione del ciclo:

1. anzitutto, una volta che si sia trovato un divisore proprio non occorre cercarne altri: quando `divisore_trovato` diventa vero, si può uscire dal ciclo;

2. secondariamente, se n è composto uno dei suoi divisori propri è certamente $\leq \sqrt{n}$: quindi la ricerca può essere tranquillamente sospesa quando `divisore` diventa $> \sqrt{n}$.

Possiamo quindi così riscrivere il programma:

```cpp
#include <iostream>
#include <cmath>
int main(){
   int n; //Numero immesso dall'utente
   int candidato; //Candidato divisore
   int soglia; //Ultimo candidato da tentare
   bool divisore_trovato; //Flag

   //Immissione del numero da parte dell'utente
   std::cout << "Inserisci un numero maggiore o uguale a 2";
   std::cin >> n;
   //Aggiungere qui eventuale trappola per assicurarsi che n>=2...

   candidato = 2; divisore_trovato = 0;
   soglia = std::min((int)ceil(sqrt(n)),n-1);
   while(candidato<=soglia && !divisore_trovato){
      if(n%candidato==0){
         divisore_trovato = 1;
      }
      candidato = candidato+1;
   }
   if(divisore_trovato){
      std::cout << "Il numero " << n << " e' composto";
   }else{
      std::cout << "Il numero " << n << " e' primo";
   }
   return 0;
}
```

La variabile `soglia` rappresenta l'*ultimo* divisore da tentare: è fondamentale non spingersi oltre $n-1$; inoltre, basta considerare divisori $\leq \sqrt{n}$. Possiamo quindi impostare come `soglia` il minimo tra $n-1$ e $\sqrt{n}$, quest'ultima approssimata ad intero per eccesso (si noti che, da un punto di vista matematico, sarebbe corretto usare anche $\sqrt{n}$ approssimata per *difetto* per la determinazione della soglia; l'uso dell'approssimazione per eccesso è prudenziale ed è in grado di garantirci contro i possibili problemi derivanti dall'imprecisione con cui il computer calcola la radice quadrata).

Si noti che, per implementare il calcolo appena descritto di `soglia`, abbiamo fatto uso delle seguenti funzioni C++:

- `std::min(quantita1, quantita2)`, che restituisce il minimo tra i due operandi (che debbono essere di un medesimo tipo numerico);
- `ceil(numero)`, che restituisce l'approssimazione per eccesso ad intero del numero fornito come argomento (dalla libreria `cmath`);
- `sqrt(numero)`, che calcola la radice quadrata del numero fornito come argomento (dalla libreria `cmath`).

Avremmo potuto omettere di introdurre una variabile `soglia`, e scrivere direttamente l'intestazione de ciclo così:

```
while(candidato<=std::min((int)ceil(sqrt(n)),n-1) && !divisore_trovato)
```

Avremmo però così costretto il programma a ricalcolare, ad ogni iterazione, l'espressione `std::min((int)ceil(sqrt(n)),n-1)`, con conseguente inutile spreco di risorse di calcolo!

7 Somme di una serie

Una *somma infinita* o *serie* è una scrittura del tipo:

$$\sum_{i=1}^{\infty} a_i = a_1 + a_2 + a_3 + a_4 + \ldots \text{(infiniti termini)}$$

La somma dei primi n termini di una serie siffatta si dice *n-esima somma parziale*; se non è elementare dare un senso matematico alla "somma della serie", le sue somme parziali, constando di un numero sempre finito di termini, si possono invece calcolare serenamente.

1. Si scriva un programma che, dato N immesso dall'utente, scriva a terminale l'N-esima somma parziale della serie:
$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \frac{1}{8} + \ldots$$
2. Si modifichi il programma, in modo che stampi a video non soltanto la somma parziale N-esima, ma anche tutte le precedenti.
3. Si provi ad eseguire il programma e ad osservare cosa si ottiene; in particolare, si confrontino le somme parziali ottenute con il numero reale $\ln(2)$.

Soluzione

Obiettivo del programma è calcolare i termini della successione:

n	Somma
0	$s_0 = 0$
1	$s_1 = 1$
2	$s_2 = 1 - 1/2$
3	$s_3 = 1 - 1/2 + 1/3$
...	...

Si noti come ogni riga della tabella possa essere calcolata dalla riga precedente:

- il nuovo n si ottiene incrementando il precedente di 1;
- la nuova somma si ottiene sommando o sottraendo alla vecchia somma il reciproco del "nuovo n".

Avremo bisogno di due variabili, `n` e `somma`, i cui valori iniziali dovranno essere quelli indicati nella prima riga della tabella. Un apposito ciclo curerà la transizione da ogni riga alla successiva, sostituendo i "vecchi" valori di `n` e di `somma` con quelli nuovi:

```
1 #include <iostream>
2 #include <iomanip>
3 #include <cmath>
4 int main(){
```

```
5     int n, N;
6     double somma;
7
8     std::cout << std::setprecision(10);
9     std::cout << std::scientific;
10    std::cout << "Quanti passi? ";
11    std::cin >> N;
12    n=0; somma = 0;
13    while(n<N){
14       n = n+1;
15       if(n%2==0){
16          somma = somma - 1.0/n;
17       }else{
18          somma = somma + 1.0/n;
19       }
20    }
21    std::cout << "La somma parziale " << n << "-esima e': " << somma;
22    std::cout << " e la differenza con ln(2) e' " << somma-log(2);
23    return 0;
24 }
```

Così scritto, il programma mostra solo l'ultima somma parziale calcolata (e la differenza tra questa e il numero ln(2)). Se si vogliono mostrare tutte le somme parziali fino all'N-esima, basta spostare dentro al ciclo i comandi di output.

L'esercizio appena svolto è un esempio di calcolo, mediante un ciclo, di una successione definita per ricorsione. Anche i due prossimi esercizi, relativi ai numeri di Fibonacci e alla scrittura dei numeri in base b, sono riconducibili allo stesso ambito. Una più dettagliata discussione di questo argomento è presentata nella soluzione dell'esercizio sui numeri di Fibonacci.

8 Fibonacci

La successione di Fibonacci è una successione di numeri interi in cui ogni termine è la somma dei due che lo precedono (e i primi due termini sono posti uguali ad 1):

$$a_0 = 1 \qquad a_1 = 1 \qquad a_n = a_{n-1} + a_{n-2}, \text{ se } n \geq 2$$

1. Si scriva un programma che stampi a terminale i primi $N + 1$ termini (da a_0 fino ad a_N) della successione, ove N è un numero intero immesso dall'utente.

2. Si scriva un programma che, dato un numero intero k immesso dall'utente da terminale, determini se k figura o meno nella successione di Fibonacci.

Successioni sulla retta

Prima di affrontare la soluzione dell'esercizio, è doveroso ragionare un poco più in generale intorno all'uso dei cicli per il calcolo dei termini di una successione. Proponiamo, come punto di partenza, la scrittura di un programma in grado di produrre i termini della successione:

$$\begin{cases} a_0 = 5 \\ a_n = 2a_{n-1} + 3n \quad \forall n \geq 1 \end{cases}$$

n	a_n
0	$a_0 = 5$
1	$a_1 = 2a_0 + 3 \cdot 1 = 2 \cdot 5 + 3 = 13$
2	$a_2 = 2a_1 + 3 \cdot 2 = 2 \cdot 13 + 6 = 32$
...	...

Avremo bisogno di due variabili: `n`, che conservi il valore dell'indice n, e `a` che memorizzi il valore del corrispondente termine a_n. Imposteremo ovviamente `0` come valore iniziale di `n`, e dunque $a_0 = 5$ come valore iniziale di `a`:

`n`: $\boxed{0}$ `a`: $\boxed{a_0}$

A questo punto, occorrerà affidare la scrittura dei termini della successione ad un ciclo: ogni iterazione avrà il compito di scrivere un termine. Immaginiamo di seguire lo svolgimento della prima di queste iterazioni: dovrà informare l'utente che a_0 vale 5; i due valori da mostrare a video sono proprio quelli immagazzinati nelle due variabili, cosicché basterà scrivere una riga di codice come:

```
std::cout << "a" << n << " vale " << a << "\n";
```

Ora che ha assolto al proprio compito, il passo iterativo, prima di concludersi, deve "preparare il terreno" per l'iterazione successiva, aggiornando debitamente i valori delle due variabili `n` ed `a`. Faremo quindi seguire le due istruzioni:

```
n = n+1;
a = 2*a+3*n;
```

con cui otterremo che, al termine della prima iterazione, il valore di `n` passi da 0 a 1, e che quello di `a` passi da $a_0 = 5$ ad $a_1 = 13$:

`n:` $\boxed{1}$ `a:` $\boxed{a_1}$

La stesura del programma è ora completata; resta solo da decidere la condizione di ciclo, che andrà scelta a seconda di quanti termini si desidera calcolare. Il programma di seguito, per esempio, scrive i primi 11 termini (da a_0 ad a_{10}) della nostra successione:

```
#include <iostream>
int main(){
   int a, n;
   n = 0;
   a = 5;
   while(n<=10){
      std::cout << "a" << n << " vale " << a << "\n";
      n = n+1;
      a = 2*a+3*n;
   }
   return 0;
}
```

Successioni nel piano

Si immagini ora di voler scrivere un programma che calcoli i termini delle due successioni a_n e b_n di seguito descritte:

$$\begin{cases} a_0 = 5; \quad b_0 = 3 & \\ a_n = a_{n-1} b_{n-1} & \forall n \geq 1 \\ b_n = a_{n-1} + b_{n-1} + n & \forall n \geq 1 \end{cases}$$

n	a_n
0	$a_0 = 5$
	$b_0 = 3$
1	$a_1 = a_0 b_0 = 5 \cdot 3 = 15$
	$b_1 = a_0 + b_0 + 1 = 5 + 3 + 1 = 9$
2	$a_2 = a_1 b_1 = 15 \cdot 9 = 135$
	$b_2 = a_1 + b_1 + 2 = 15 + 9 + 2 = 26$
...	...

Si osservi che a_n e b_n non possono essere calcolate indipendentemente l'una dall'altra: per calcolare a_n occorre infatti conoscere il termine precedente sia di a che di b, e lo stesso dicasi per b_n. Piuttosto che di due successioni, è quindi più giusto dire che si tratti di una successione singola, i cui termini non sono singoli numeri (cioè punti della retta) ma coppie di numeri (cioè punti del piano).

Il primo tentativo che intraprenderemo consiste semplicemente nel riciclare il programma già scritto, adattandolo nel modo ovvio:

```
1  #include <iostream>
2  int main(){
3     int a, b, n;
4     n = 0;
5     a = 5; b = 3;
6     while(n<=3){
7        std::cout << "a" << n << " vale " << a << " e "
8                  << "b" << n << " vale " << b << "\n";
9        n = n+1;
10       a = a*b;
11       b = a+b+n;
12    }
13    return 0;
14 }
```

Il programma scritto dovrebbe mostrare a video i primi 4 termini della successione, da (a_0, b_0) fino ad (a_3, b_3). Contiene però un errore: per capirne la ragione, è opportuno seguire, passo per passo, il suo svolgimento...

Le variabili `n`, `a` e `b`, una volta dichiarate, vengono inizializzate, rispettivamente, con i valori 0, $a_0 = 5$ e $b_0 = 3$:

`n:` $\boxed{0}$ `a:` $\boxed{a_0}$ `b:` $\boxed{b_0}$

Prende allora avvio la prima iterazione del ciclo: il programma scrive correttamente a terminale che "`a0 vale 5 e b0 vale 3`". A questo punto, intervengono le istruzioni alla riga 9, 10 e 11 ad aggiornare i valori delle tre variabili in vista dell'iterazione successiva. Eseguita la riga 9, `n` passa correttamente a valere 1:

`n:` $\boxed{1}$ `a:` $\boxed{a_0}$ `b:` $\boxed{b_0}$

e quando esegue la riga 10, il computer calcola `a*b`, ottenendo $a_0 b_0 = 5 \cdot 3 = 15$, cioè a_1, e imposta correttamente tale risultato come nuovo valore di `a`:

`n:` $\boxed{1}$ `a:` $\boxed{a_1}$ `b:` $\boxed{b_0}$

Raggiunta la riga 11, il computer dovrebbe determinare b_1 e impostarlo come nuovo valore di `b`; ma, quanto calcola `a+b+n`, ottiene $a_1 + b_0 + 1$, e non b_1, che è invece dato da $a_0 + b_0 + 1$! Per eseguire il calcolo in modo corretto, il computer avrebbe bisogno del vecchio valore a_0 di `a`, il quale però è stato ormai sacrificato per far posto ad a_1!

La soluzione è introdurre una **variabile temporanea**, che chiameremo `tmp`: prima di cambiare il valore di `a` (cioè prima della riga 10), per non condannare improvvidamente a_0 all'oblio provvederemo a ricopiarlo dentro a `tmp`. Potremo poi cambiare liberamente il valore di `a`, perché a_0 sopravviverà in ogni caso dentro a `tmp`, e sarà quindi a nostra disposizione quando ci troveremo a dover calcolare calcolare b_1.

In definitiva, un modo corretto di scrivere il programma è questo:

```cpp
#include <iostream>
int main(){
   int a, b, tmp, n;
   n = 0;
   a = 5; b = 3;
   while(n<=3){
      std::cout << "a" << n << " vale " << a << " e "
                << "b" << n << " vale " << b << "\n";
      n = n+1;
      tmp = a;
      a = a*b;
      b = tmp+b+n;
   }
   return 0;
}
```

Una soluzione un poco più verbosa, ma anche più simmetrica e leggibile, è la seguente, che fa uso di due variabili temporanee in cui i "vecchi" valori di `a` e di `b` vengono messi al sicuro e preservati dalla sovrascrittura:

```cpp
#include <iostream>
int main(){
   int a, b, a_vecchio, b_vecchio, n;
   n = 0;
   a = 5; b = 3;
   while(n<=3){
      std::cout << "a" << n << " vale " << a << " e "
                << "b" << n << " vale " << b << "\n";
      n = n+1;
      a_vecchio = a;
      b_vecchio = b;
      a = a_vecchio*b_vecchio;
      b = a_vecchio+b_vecchio+n;
   }
   return 0;
}
```

La successione di Fibonacci (soluzione punto 1)

Veniamo ora alla successione di Fibonacci. La logica dei programmi che abbiamo scritto nei paragrafi precedenti si adatta al calcolo di ogni successione del tipo:

$$\begin{cases} \mathbf{x}_0 \text{ fissato} \\ \mathbf{x}_n = \mathbf{F}(\mathbf{x}_{n-1}, n) \quad \forall n \geq 1 \end{cases} \tag{$\star$}$$

dove il termine generale $\mathbf{x}_n \in \mathbb{R}^d$ è un punto dello spazio d-dimensionale, e $\mathbf{F} : \mathbb{R}^d \times \mathbb{N} \to \mathbb{R}^d$ è la funzione che esprime il nuovo termine in funzione del precedente.

La successione di Fibonacci non sembra però, a prima vista, presentarsi in tale forma: i termini "fissati" sono due, al posto che uno soltanto; inoltre, il termine n-esimo, per essere calcolato, richiede la conoscenza non solo del precedente, cioè

l'$(n-1)$esimo, ma anche di quello ancora prima, cioè l'$(n-2)$esimo. Un semplice trucco, che ora illustreremo, ci consentirà però di riportare anche la successione di Fibonacci alla forma $\star$. Sia a_n la successione di Fibonacci:

n	0	1	2	3	4	5	6	7	8	...
a_n	1	1	2	3	5	8	13	21	34	...

Le affiancheremo una seconda successione b_n, ottenuta eliminando il termine di testa e facendo conseguentemente slittare indietro tutti gli altri: in altre parole, $b_n = a_{n+1}\ \forall n \geq 0$, o, il che è lo stesso, $b_{n-1} = a_n\ \forall n \geq 1$:

n	0	1	2	3	4	5	6	7	8	...
a_n	1	1	2	3	5	8	13	21	34	...
b_n	1	2	3	5	8	13	21	34	55	...

È semplice accorgersi che, per la "successione nel piano" (a_n, b_n), il termine n-esimo può essere espresso come funzione del solo termine $(n-1)$-esimo: infatti, $\forall n \geq 1$,

$$\begin{cases} a_n = b_{n-1} \\ b_n = a_{n-1} + b_{n-1} \end{cases}$$

Anche la condizione iniziale $a_0 = 1, a_1 = 1$, che coinvolge due termini, può essere equivalentemente riscritta facendo riferimento al solo termine di posizione 0 della successione doppia: $a_0 = 1, b_0 = 1$.

La nostra successione di Fibonacci "raddoppiata" può essere quindi presentata come segue:

$$\begin{cases} a_0 = 1, b_0 = 1 & \\ a_n = b_{n-1} & \forall n \geq 1 \\ b_n = a_{n-1} + b_{n-1} & \forall n \geq 1 \end{cases}$$

e rientra quindi nella forma $\star$, cosicché il calcolo della successione di Fibonacci può essere serenamente affidato ad un programma del tutto analogo all'ultimo che abbiamo scritto:

```
#include <iostream>
int main(){
   int a, b, a_vecchio, b_vecchio, n, N;
   std::cout << "Fino a che termine devo arrivare? ";
   std::cin >> N;
   n = 0;
   a = 1; b = 1;
   while(n<=N){
      std::cout << "Il " << n << "esimo numero di Fibonacci "
                << "vale " << a << "\n";
      n = n+1;
      a_vecchio = a;
      b_vecchio = b;
      a = b_vecchio;
      b = a_vecchio+b_vecchio;
   }
   return 0;
}
```

Soluzione punto 2

Scriviamo ora un programma che verifichi se un numero k immesso dall'utente appartenga o meno alla successione di Fibonacci. L'idea è quella di scorrere uno dopo l'altro i numeri di Fibonacci, controllando, per ognuno di essi, se coincide o meno con k. La ricerca dovrà essere arrestata quando, alternativamente:

- il termine in esame della successione di Fibonacci risulti uguale a k;
- si sia ormai raggiunto un termine della successione di Fibonacci maggiore di k, e sarebbe quindi vano continuare la ricerca.

L'esito della ricerca sarà memorizzato in un flag `trovato`:

```
#include <iostream>
int main(){
   int a, b, a_vecchio, b_vecchio, k;
   bool trovato;
   std::cout << "Inserisci un numero: ";
   std::cin >> k;
   trovato = 0;
   a = 1; b = 1;
   while(!trovato && a<=k){
      if(a==k){
         trovato = 1;
      }
      a_vecchio = a;
      b_vecchio = b;
      a = b_vecchio;
      b = a_vecchio+b_vecchio;
   }

   if(trovato){
      std::cout << k << " e' un numero di Fibonacci" << "\n";
   }else{
      std::cout << k << " non e' un numero di Fibonacci" << "\n";
   }

   return 0;
}
```

Si noti come, nel codice, sia stata omessa la variabile contatore `n`, in quanto superflua (sarebbe però necessario conservarla nel caso in cui si volesse conoscere, quando k è un numero di Fibonacci, la sua posizione nella successione).

Un'elegante soluzione alternativa (senza uso di flag) è la seguente:

```
#include <iostream>
int main(){
   int a, b, a_vecchio, b_vecchio, k;
   std::cout << "Inserisci un numero: ";
   std::cin >> k;
   a = 1; b = 1;
   while(a<k){
      a_vecchio = a;
      b_vecchio = b;
      a = b_vecchio;
      b = a_vecchio+b_vecchio;
   }

   if(a==k){
      std::cout << k << " e' un numero di Fibonacci" << "\n";
   }else{
      std::cout << k << " non e' un numero di Fibonacci" << "\n";
   }

   return 0;
}
```

Al termine del ciclo, nella variabile `a` troviamo memorizzato il primo numero di Fibonacci che falsifica la condizione `a<k`, cioè il minimo numero di Fibonacci maggiore o uguale a k, ed è a questo punto tautologica l'osservazione che:

$$k \text{ è di Fibonacci} \iff k = \min\{\text{numeri di Fibonacci} \geq k\}$$

9 Base b

Si scriva un programma che, dato un numero naturale $n \geq 1$ e una base $b \geq 2$ immessi dall'utente, scriva a video le cifre del numero in base b, seguendo l'algoritmo presentato all'inizio del corso (non è un problema se il programma scrive le cifre in ordine inverso...).

Soluzione

Rammentiamo l'algoritmo presentato all'inizio del corso: per scrivere un numero $n \in \mathbb{N}$ in base b, si calcolano il quoziente intero e il resto della sua divisione per b: il resto fornisce una prima cifra; il quoziente va assoggettato al medesimo procedimento cui è stato sottoposto n, ... e così si procede, fintanto che il quoziente non si annulli:

```
#include <iostream>
int main(){
   int n, b;
   std::cout << "Inserisci un numero naturale: ";
   std::cin >> n;
   std::cout << "Inserisci la base: ";
   std::cin >> b;
   std::cout << "Ecco " << n << " scritto in base " << b << ":\n";
   while(n>0){
      //Scrivo la cifra
      std::cout << n%b << " ";

      //Aggiorno n, sostituendolo, in vista della prossima iterazione,
      //con il suo quoziente intero nella divisione per b
      n = n/b;
   }
   return 0;
}
```

Così facendo, il numero viene però scritto "al contrario". Se si vuole ovviare a questo inconveniente, bisogna memorizzare le cifre via via calcolate in un array al posto che stamparle a video, e poi, con un successivo ciclo, scriverle a terminale riavvolgendolo:

```cpp
#include <iostream>
int main(){
   int n, b, i;
   int cifre[32];
   std::cout << "Inserisci un numero naturale: ";
   std::cin >> n;
   std::cout << "Inserisci la base: ";
   std::cin >> b;
   std::cout << "Ecco " << n << " scritto in base " << b << ":\n";

   i = 0;
   while(n!=0){
      //Memorizzo la cifra nell'array
      cifre[i] = n%b;

      //Aggiorno n, sostituendolo, in vista della prossima iterazione,
      //con il suo quoziente intero nella divisione per b
      n = n/b;

      //Incremento il contatore i, di modo che alla prossima iterazione
      //il ciclo scriva nella successiva casella dell'array
      i = i+1;
   }

   //Terminato il ciclo di scrittura, correggo i in modo che contenga
   //l'indice dell'ultima casella scritta
   i = i-1;

   //Scorro ora, all'indietro, tutte le entrate dell'array

   while(i>=0){
      //Scrivo la cifra presente nell'i-esima casella dell'array
      std::cout << cifre[i] << " ";

      //Faccio un passo indietro in vista della prossima iterazione
      i = i-1;
   }
   return 0;
}
```

La lunghezza 32 impostata per l'array al momento della dichiarazione è prudenziale, ed è sempre sufficiente, quale che sia la base, fintanto che n sia un intero a 32bit.

5 Funzioni

1 Massimo comune divisore

1.1 Prefazione

Dati due numeri $a, b \in \mathbb{N}$, si consideri la successione (a_n, b_n) di coppie di numeri naturali descritta di seguito per ricorsione:

$$\begin{cases} a_0 = a,\ b_0 = b & \\ a_n = b_{n-1} & \forall n \geq 1 \\ b_n = \text{resto della divisione di } a_{n-1} \text{ per } b_{n-1} & \forall n \geq 1 \end{cases}$$

Non è difficile dimostrare che b_n è strettamente decrescente (cioè che $b_0 > b_1 > b_2 > \dots$); in particolare, dopo un certo numero finito di passi, b_n diventa necessariamente 0. Quando b_n raggiunge 0, la successione (a_n, b_n) **si arresta**: per ottenere un termine ulteriore bisognerebbe infatti calcolare il resto di una divisione euclidea per 0, cosa non possibile.

Si può dimostrare la seguente

Proposizione (Euclide)**.** *Sia (a_N, b_N), con $b_N = 0$, l'ultimo termine della successione suindicata. Allora a_N è il massimo comune divisore tra a e b.*

Per esempio: se $a = 240$ e $b = 90$, la successione è fatta in questo modo:

$n = 0$	$n = 1$	$n = 2$	$n = 3$	
$(240, 90)$	$(90, 60)$	$(60, 30)$	$(30, \mathbf{0})$	**STOP**

e 30 è effettivamente il massimo comune divisore tra 240 e 90.

1.2 Consegna

1. Si scriva una funzione `mcd` che, accettati come argomenti due numeri naturali (tipo `int`), ne restituisca il massimo comune divisore (tipo `int`), calcolato utilizzando l'algoritmo descritto nella prefazione.

   Si scriva inoltre un `main` che consenta all'utente di interfacciarsi con la funzione (concluso lo svolgimento di questo punto dell'esercizio, rinominarlo `main1`).

2. Si scriva un `main` che, chiesto all'utente un numero naturale N, elenchi a terminale gli interi compresi tra 0 ed N che sono coprimi[a] con N, scriva quanti sono in tutto e indichi quanto vale la loro somma. Concluso lo svolgimento di questo punto dell'esercizio, rinominare il `main` che si è scritto `main2`.

3. Si scriva una funzione `lancio` che accetti come argomento un numero naturale M (che si supporrà, senza verificarlo, ≥ 1), estragga a sorte due numeri naturali compresi tra 1 ed M[b] e restituisca il valore di verità `VERO` (cioè 1) se tali numeri sono coprimi, `FALSO` (cioè 0) altrimenti.

[a] Due numeri naturali si dicono *coprimi* quando il loro massimo comune divisore è 1

[b] Per generare numeri interi casuali, si faccia uso della funzione `fiore::irand` del pacchetto `fiore_random.h`, che accetta come parametri due estremi interi a, b e restituisce un **intero** casuale nell'intervallo $[a, b]$

Si scriva un `main` che utilizzi la funzione `lancio` per generare N coppie di numeri interi casuali compresi tra 1 e M (con N, M immessi da terminale), e che comunichi a terminale la frequenza relativa delle coppie di interi coprimi. Cosa si osserva per N ed M grandi?

4. Nella prefazione si afferma che b_n è strettamente decrescente. Si dimostri (in non più di due righe!) tale affermazione.

1.3 Soluzione

Punto 1

```
#include <iostream>
int mcd(int a, int b){
   int a_vecchio, b_vecchio;
   while(b>0){
      //Salvo gli attuali valori di a e b
      //nelle variabili "temoranee" a_vecchio e b_vecchio
      a_vecchio = a;
      b_vecchio = b;

      //Aggiorno a e b secondo la legge prescritta
      a = b_vecchio;
      b = a_vecchio%b_vecchio;
   }
   return a;
}

int main1(){
   int a, b, c;
   std::cout << "Inserisci due numeri naturali ";
   std::cin >> a >> b;
   c = mcd(a,b);
   std::cout << "Il loro MCD e' " << c;
   return 0;
}
```

Punto 2

```
//Omissis

int main2(){
   int N, n, conto, somma;
   std::cout << "Inserisci N ";
   std::cin >> N;
   std::cout << "I num tra 0 e " << N << " coprimi con " << N << " sono: ";
   n = 0; conto = 0; somma = 0;
   while(n<=N){
```

```
10      if(mcd(N,n)==1){
11         std::cout << n << " ";
12         conto = conto+1;
13         somma = somma+n;
14      }
15      n = n+1;
16   }
17   std::cout << "\n" << "Sono in tutto " << conto;
18   std::cout << " e la loro somma e' " << somma;
19   return 0;
20 }
```

Punto 3

```
#include <iostream>
#include "fiore_random.h"
//Omissis

bool lancio(int M){
   int a, b;
   a = fiore::irand(1,M);
   b = fiore::irand(1,M);
   return mcd(a,b)==1;
}
int main(){
   int N, M, i, successi;
   double freq_coprimi;
   std::cout << "Numero N di coppie: ";
   std::cin >> N;
   std::cout << "Numero M fin cui arrivare: ";
   std::cin >> M;
   i=0; successi = 0;
   while(i<N){
      if(lancio(M)){
         successi=successi+1;
      }
      i = i+1;
   }
   freq_coprimi = successi/(double)N;
   std::cout << "Freq. rel. coppie di numeri coprimi: " << freq_coprimi;
   return 0;
}
```

Per N ed M grandi, la frequenza relativa tende verso un limite $0 < l < 1$ (che si può dimostrare essere uguale a $6/\pi^2$).

Punto 4

Il resto di una divisione euclidea tra numeri naturali è sempre minore del divisore. Dunque, $\forall n \geq 1$, $b_n < b_{n-1}$, il che significa che la successione b_n è monotona strettamente decrescente.

2 Essere potenza di un primo

1. Si scriva una funzione `minimo_div` che, accettando come parametro un numero intero n che si suppone, senza necessità di verificarlo, essere ≥ 2, restituisce il suo più piccolo divisore maggiore di 1.

   Si scriva un `main` "temporaneo" che chiami la funzione, per testarla (concluso lo svolgimento di questo punto dell'esercizio, rinominarlo `main1`).

2. Si descriva, da un punto di vista teorico, l'insieme $S \subseteq \mathbb{N}$ dei possibili valori che la funzione `minimo_div` può restituire, giustificando adeguatamente la risposta.

3. Si scriva una funzione `potenza_primo` che, dato un parametro intero n, e assumendo, senza verificarlo, $n \geq 0$, restituisca `VERO` (cioè 1) se n è una potenza di un qualche numero naturale primo, `FALSO` (cioè 0) altrimenti. La funzione deve restituire il risultato corretto anche nei due casi limite $n = 0$ e $n = 1$.

   Si scriva un `main` che consenta all'utente di interfacciarsi con la funzione `potenza_primo`.

2.1 Soluzione

Punto 1

```
#include <iostream>
int minimo_div(int n){
   int candidato_div;
   candidato_div = 2;
   while(1){
      if(n%candidato_div==0){
         return candidato_div;
      }
      candidato_div = candidato_div+1;
   }
}
int main1(){
   int n;
   std::cout << "Inserisci n, maggiore o uguale a 2 ";
   std::cin >> n;
   std::cout << "Il minimo divisore di " << n << " e' " << minimo_div(n);
}
```

Punto 2

S è l'insieme dei numeri primi. Segue la dimostrazione.

S è costituito di soli numeri primi Infatti, sia $n \in \mathbb{N}$, $n \geq 2$, e sia d il minimo divisore di n (escluso 1). Se d, per assurdo, non fosse primo, allora ammetterebbe una qualche decomposizione non banale $d = d_1 d_2$, essendo $d_1, d_2 \in \mathbb{N}$, $1 < d_1 < n$, $1 < d_2 < n$. d_1 e d_2 sarebbero divisori di n maggiori di 1 e strettamente minori di d, il che contraddice alla supposta minimalità di d.

Ogni numero primo sta in S Infatti, se n è un qualunque primo, il minimo divisore di n (escluso 1) è proprio n stesso, e dunque $n \in S$.

Punto 3

L'osservazione cruciale è che, se n è potenza di un primo, allora tale primo è il suo minimo divisore... Detto altrimenti, dato n, l'unico primo di cui n è candidato ad essere potenza è proprio il minimo divisore di n (che denoteremo con d).

Una possibile implementazione di `potenza_primo` è questa, in cui si calcola la minima potenza di d maggiore o uguale ad n, e si vede se essa coincide o meno con n stesso:

```
bool potenza_primo(int n){
   int d; //Minimo divisore di n
   int potenza_d; //Scorre le potenze di d
   if(n==0){
      return false;
   }
   if(n==1){
      return true;
   }
   d = minimo_div(n); potenza_d = 1;
   while(potenza_d < n){
      potenza_d = potenza_d * d;
   }
   if(potenza_d==n){
      return true;
   }else{
      return false;
   }
}
```

Una diversa implementazione di `potenza_primo` si può realizzare dividendo ricorsivamente, fin dove è possibile, n per d, per poi controllare se ciò che rimane è o non è uguale ad 1.

```
bool potenza_primo(int n){
   int d; //Minimo divisore di n
   if(n==0){
      return false;
   }
   if(n==1){
      return true;
   }
   d = minimo_div(n);
   while(n%d==0){
      n=n/d;
   }
   if(n==1){
      return true;
   }else{
      return false;
   }
}
```

Si lascia al lettore la scrittura di un `main` che consenta all'utente di interfacciarsi con la funzione.

6 Calcolo delle aree

1 Metodi Monte Carlo

Sia C un cerchio di raggio 1, centrato nell'origine degli assi, inscritto entro un quadrato Q di semilato uguale a 1, centrato anch'esso nell'origine con lati paralleli agli assi. Si intende stimare l'area del cerchio C in questo modo: si scelgono N punti casuali del quadrato (ci riferiremo a tali punti designandoli come *tentativi* o *lanci*) e si valuta quanti di essi cadono nel cerchio (cioè quanti dei *lanci* sono *successi*): la frequenza relativa di successo (numero di successi in rapporto al numero di tentativi) converge al rapporto di aree $\text{Area}(C)/\text{Area}(Q)$ al crescere del numero N di lanci.

Il programma

1. Si scriva anzitutto una funzione `lancio` che scelga un punto casuale del quadrato e restituisca `VERO` se esso appartiene al cerchio, `FALSO` in caso contrario[(a)].

2. Si scriva poi una funzione `freq` che, accettando come argomento un numero intero `N`, esegua `N` lanci e restituisca la frequenza relativa di successo.

3. Si scriva infine un `main` che comunichi all'utente una stima dell'area del cerchio C ottenuta mediante N lanci, con N immesso dall'utente. Si esegua il programma più volte, immettendo vari valori di N, e si tenti di osservare, sperimentalmente, l'andamento dell'errore in ragione di N.

Altre metriche Se $\alpha \in \mathbb{R}^+$, la *α-distanza* tra due punti A e B del piano si calcola come: $d_\alpha(A,B) = (|x_A - x_B|^\alpha + |y_A - y_B|^\alpha)^{1/\alpha}$. La *palla unitaria centrata nell'origine per la distanza d_α* (o, più semplicemente, *la palla di d_α*) è definita come il luogo dei punti che distano meno di 1 dall'origine. La distanza che si ottiene per $\alpha = 2$ è l'usuale distanza euclidea, e la sua palla è il cerchio di raggio 1 centrato nell'origine.

4. Si generalizzi il programma scritto precedentemente in modo che consenta di stimare l'area della d_α-palla. In particolare, il `main` dovrà chiedere all'utente un valore di α, che andrà poi trasmesso alla funzione `freq`, la quale a sua volta lo dovrà comunicare a `lancio`...

5. Si scriva un nuovo `main` (rinominare il precedente in `main1`) che studi l'andamento del rapporto di aree $\text{Area}(B_\alpha)/\text{Area}(Q)$, dove B_α è la d_α-palla, al variare di α (si prenda $\alpha \in [0,4]$, si usi un passo $\Delta\alpha = 0.05$, e si stimi ogni area eseguendo $N = 5 \cdot 10^4$ lanci). Si commenti il risultato ottenuto.

Altre dimensioni

6. Si generalizzino tutte le funzioni scritte in modo che possano affrontare i problemi loro affidati in uno spazio euclideo di dimensione d qualsiasi (per intendersi: se, per esempio, $d = 3$, l'area diverrà un volume, il quadrato un cubo, il cerchio una palla piena, ...)

[(a)] Per generare un numero casuale `double` compreso tra due estremi a e b, si usi la funzione `fiore::drand(a,b)`, definita nella libreria "fiore_random.h". Tale libreria va preventivamente scaricata dal sito del corso, salvata nella medesima cartella ove si trova il sorgente `.cpp` del programma e inclusa mediante la direttiva di compilazione `#include "fiore_random.h"`

1.1 Soluzione

Punto 1

La prima cosa da fare, per scrivere una funzione, è deciderne la *segnatura*, il che equivale a domandarsi:

- quali dati la funzione deve esigere da chi la invoca;
- quale tipo di risultato la funzione deve restituire a chi l'ha invocata come "prodotto finale" del proprio lavoro.

Nel caso di specie, appare evidente che:

- la funzione `lancio` non ha bisogno di alcuna informazione in ingresso per poter assolvere al proprio compito (che consiste nel generare un punto casuale del quadrato di centro O e semilato 1 e stabilire se appartenga o meno al cerchio di centro O e raggio 1);
- ciò che la funzione `lancio` deve restituire a chi la chiama, stando alle indicazioni fornite nel testo dell'esercizio, è un valore di verità (tipo `bool`)

La definizione della funzione `lancio` avrà dunque quest'aspetto:

```
bool lancio(){
   //Corpo della funzione
}
```

Rimane ora da scrivere il *corpo* della funzione. `lancio` deve anzitutto generare un punto casuale del quadrato, cioè una coppia di coordinate (x, y), con $x, y \in [-1, 1]$. Dichiareremo allora due variabili `double` (che saranno visibili *solo* a `lancio` nel corso della sua esecuzione) e conferiremo loro due valori casuali:

```
bool lancio(){
   double x, y;
   x = fiore::drand(-1,1);
   y = fiore::drand(-1,1);
   //Continua...
}
```

La funzione `lancio` deve ora decidere se (x, y) è o meno un punto del cerchio; a questo scopo, giova ricordare che l'equazione cartesiana del cerchio unitario di centro l'origine è $x^2 + y^2 \leq 1$. Allora:

```
bool lancio(){
   double x, y;
   x = fiore::drand(-1,1);
   y = fiore::drand(-1,1);
   if(x*x+y*y<=1){
      return 1;
   }else{
      return 0;
   }
}
```

o ancor più brevemente:

```
bool lancio(){
   double x, y;
   x = fiore::drand(-1,1);
   y = fiore::drand(-1,1);
   return x*x+y*y <= 1;
}
```

Per testare la funzione, possiamo scrivere un `main` "temporaneo" che si occupi di invocarla: il nostro programma avrà così questo aspetto:

```
1  #include <iostream>
2  #include "fiore_random.h"
3  bool lancio(){
4     double x, y;
5     x = fiore::drand(-1,1);
6     y = fiore::drand(-1,1);
7     return x*x+y*y <= 1;
8  }
9  int main(){
10    std::cout << "Esito di un lancio ";
11    std::cout << lancio() << "\n";
12    std::cout << "Esito di un secondo lancio ";
13    std::cout << lancio() << "\n";
14    return 0;
15 }
```

Si noti l'ordine in cui le due funzioni sono definite nel codice: prima è definita `lancio`, poi `main`. La ragione è che il compilatore, per motivi tecnici, richiede che nel file sorgente la definizione di ogni funzione preceda sempre le sue chiamate (qui, la definizione di `lancio` occupa le righe da 3 a 8, e le sue chiamate si trovano alle righe 11 e 13, dunque il criterio è rispettato).

Punto 2

Siamo ora pronti ad introdurre una ulteriore funzione: `freq`. Stabiliamo, per prima cosa, la sua segnatura:

- le specifiche fornite nel testo dell'esercizio richiedono esplicitamente che la funzione `freq` *riceva dal chiamante* il numero N dei lanci da eseguire;
- sempre le specifiche precisano che `freq` deve restituire al chiamante una frequenza relativa, che sarà evidentemente un numero reale (tipo `double`).

Si avrà dunque:

```
1 double freq(int N){
2    //Corpo della funzione
3 }
```

Si osservi che per *invocare* la funzione `freq`, il chiamante dovrà fornire un numero di lanci (così, `freq(20)` è una valida chiamata alla funzione, mentre `freq()` non lo è!) e la funzione `freq` avrà a disposizione tale dato (immagazzinato in una variabile di nome `N` locale alla funzione) nel corso della propria esecuzione.

Veniamo alla scrittura del corpo della funzione: `freq` deve chiamare `lancio` N volte, e tener conto del numero di successi conseguiti, per poi restituire il rapporto tra il numero di successi e il numero di lanci:

```
double freq(int N){
   int i, successi;
   i = 0; successi = 0;
   while(i<N){
      if(lancio()){
         successi = successi+1;
      }
      i = i+1;
   }
   return successi/(double)N;
}
```

Se aggiungiamo un `main` "temporaneo" che chiami, per testarla, la funzione `freq`, il programma assumerà questo aspetto:

```
#include <iostream>
#include "fiore_random.h"
bool lancio(){
   double x, y;
   x = fiore::drand(-1,1);
   y = fiore::drand(-1,1);
   return x*x+y*y <= 1;
}
double freq(int N){
   int i, successi;
   i = 0; successi = 0;
   while(i<N){
      if(lancio()){
         successi = successi+1;
      }
      i = i+1;
   }
   return successi/(double)N;
}
int main(){
   std::cout << "Frequenza di successo su 1000 lanci: ";
   std::cout << freq(1000) << "\n";
   return 0;
}
```

Punto 3

Scriviamo ora un `main` che curi l'interazione con l'utente: leggerà da terminale il numero di lanci scelto dall'utente, eseguirà `freq` informandola debitamente sul numero di lanci da effettuare, e comunicherà a terminale, all'utente, la stima che ne risulta dell'area del cerchio e la sua differenza con il valore teorico π:

```
1  #define _USE_MATH_DEFINES
2  #include <iostream>
3  #include <iomanip>
4  #include <cmath>
5  #include "fiore_random.h"
6  bool lancio(){
7     double x, y;
8     x = fiore::drand(-1,1);
9     y = fiore::drand(-1,1);
10    return x*x+y*y <= 1;
11 }
12 double freq(int N){
13    int i, successi;
14    i = 0; successi = 0;
15    while(i<N){
16       if(lancio()){
17          successi = successi+1;
18       }
19       i = i+1;
20    }
21    return successi/(double)N;
22 }
23 int main(){
24    int num_lanci;
25    double stima;
26
27    std::cout << std::scientific;
28    std::cout << std::setprecision(5);
29
30    std::cout << "Numero dei lanci da eseguire: ";
31    std::cin >> num_lanci;
32
33    stima = 4*freq(num_lanci);
34    std::cout << "Stima area cerchio:\t" << stima << "\n";
35    std::cout << "Errore assoluto:\t" << (stima-M_PI);
36    return 0;
37 }
```

È di fondamentale importanza l'osservazione che la funzione `freq`, quando viene chiamata dal `main`, *non può* accedere alla variabile `num_lanci`: ciò che `freq` ha a disposizione, quando inizia la sua esecuzione, è la sola variabile `N`, che al momento della chiamata `freq(num_lanci)` (riga 33), viene valorizzata con una *copia* del valore della variabile `num_lanci`.

Ov'anche si fosse usato, per `num_lanci` e per `N`, un medesimo nome (cioè se le si fosse, per esempio, chiamate entrambe `N`), ugualmente sarebbero rimaste due variabili completamente distinte, l'una visibile al solo `main`, l'altra visibile alla sola `freq`, creata ad ogni chiamata di `freq` con dentro il numero intero passato dal chiamante e distrutta al termine di ogni esecuzione di `freq`.

Punto 4

Per generalizzare il programma in modo che possa lavorare con la palla della distanza d_α, è necessario modificare la funzione `lancio`, nel cui codice la condizione $x^2+y^2 \leq 1$ andrà sostituita con $|x|^\alpha + |y|^\alpha \leq 1$. Per poter svolgere il proprio compito, `lancio` ha bisogno conoscere il valore di α, che dovrà esigere dal chiamante: si rende quindi necessaria una modifica della *segnatura* della funzione...

PRIMA

```
bool lancio(){
   //Codice
   //Codice
   //Codice
}
```

DOPO

```
bool lancio(double alpha){
   //Codice in cui posso usare il
   //valore di alpha
   //comunicato dal chiamante
}
```

Dopo questa modifica, la funzione `freq` non può più chiamare `lancio` mediante l'invocazione `lancio()`: per chiamare `lancio` è infatti ora obbligatorio fornire un valore per il parametro `alpha`. La funzione `freq` ha quindi *a sua volta* bisogno, per poter lavorare, di conoscere α, e dovrà quindi *a sua volta* esigere il valore di α dal chiamante:

PRIMA

```
double freq(int N){
   //Omissis
   lancio();
   //Omissis
}
```

DOPO

```
double freq(int N, double alpha){
   //Omissis
   lancio(alpha);
   //Omissis
}
```

Ora che la segnatura di `freq` è cambiata, il `main` non può più chiamarla mediante l'invocazione `freq(num_lanci)` che abbiamo adoperato nella precedente versione del programma: il `main` dovrà infatti fornire a `freq`, oltre all'indicazione del numero di lanci, anche il valore del parametro `alpha`; l'una come l'altra informazione dovranno esser domandate dal `main` all'utente, immagazzinate in opportune variabili e passate a `freq`. Il programma avrà infine questo aspetto:

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <iomanip>
#include <cmath>
#include "fiore_random.h"
bool lancio(double alpha){
   double x, y;
   x = fiore::drand(-1,1);
   y = fiore::drand(-1,1);
   return pow(fabs(x),alpha)+pow(fabs(y),alpha) <= 1;
}
double freq(int N, double alpha){
   int i, successi;
   i = 0; successi = 0;
   while(i<N){
      if(lancio(alpha)){
         successi = successi+1;
      }
      i = i+1;
   }
   return successi/(double)N;
}
int main(){
   int num_lanci; double alpha;
   double stima;

   std::cout << std::scientific;
   std::cout << std::setprecision(5);

   std::cout << "Numero dei lanci da eseguire: ";
   std::cin >> num_lanci;
   std::cout << "Alpha: ";
   std::cin >> alpha;

   stima = 4*freq(num_lanci,alpha);
   std::cout << "Stima area palla:\t" << stima << "\n";
   return 0;
}
```

Riepilogando, il valore di α viene chiesto all'utente dal `main`; il `main` lo passa a `freq`; `freq` lo trasmette a `lancio`.

Si presti particolare attenzione alla logica secondo la quale la divisione di compiti tra le tre funzioni è stata operata: il `main` cura **in esclusiva** l'interazione con l'utente; viceversa, le due funzioni "ausiliarie" `freq` e `lancio` sono **mute**; infatti:

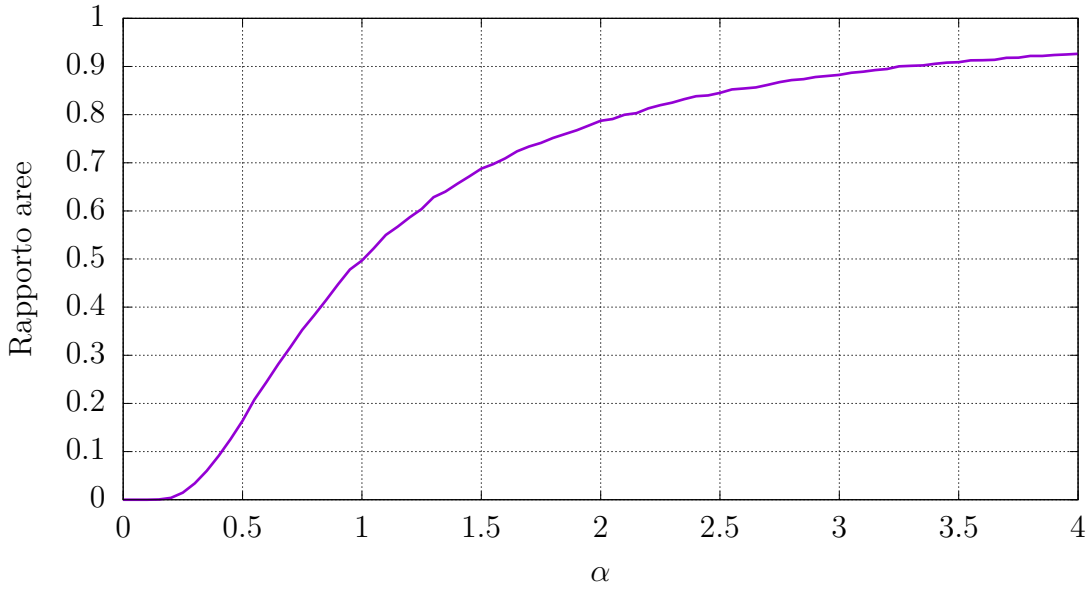
- piuttosto che interpellare l'utente tramite il terminale, pretendono di ricevere dal chiamante tutte le informazioni di cui hanno bisogno per lavorare;
- piuttosto che comunicare a terminale l'esito delle proprie elaborazioni, lo restituiscono, mediante un `return`, al chiamante.

Punto 5

Il nuovo `main` che il punto 5 dell'esercizio chiede di scrivere deve chiamare la funzione `freq` non una, ma molte volte, utilizzando valori di `alpha` via via più grandi, che sarà necessario scorrere mediante un apposito ciclo:

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <iomanip>
#include <cmath>
#include "fiore_random.h"
bool lancio(double alpha){
   //Omissis
}
double freq(int N, double alpha){
   //Omissis
}
int main1(){
   //Omissis
}
int main(){
   double alpha, alpha_min, alpha_max, delta_alpha;
   int num_lanci;

   alpha_min = 0; alpha_max = 4; delta_alpha = 0.05; num_lanci = 50000;

   std::cout << std::scientific;
   std::cout << std::setprecision(5);
   std::cout << "alpha" << "\t" << "stima rapporto aree" << "\n";

   alpha = alpha_min;
   while(alpha <= alpha_max){
      std::cout << alpha << "\t" << freq(num_lanci, alpha) << "\n";
      alpha = alpha+delta_alpha;
   }
   return 0;
}
```

Riportando su un grafico i risultati stampati a terminale dal programma, si ottiene una curva di questo genere:



Punto 6

Così come s'è già fatto per α, il numero di dimensioni dovrà essere domandato dal `main` all'utente; dovrà poi essere passato a `freq`, che dovrà comunicarlo a `lancio`. Una volta adattate adeguatamente segnature e chiamate alle funzioni, ciò che rimane da fare è modificare il codice di `lancio` in modo che sia in grado di scegliere un punto casuale dell'ipercubo d-dimensionale, e di decidere sulla sua appartenenza alla d_α-palla.

Generare il punto significa generare d numeri reali casuali compresi tra -1 e 1, che indicheremo con $x_1, \ldots, x_d$. L'equazione della palla, in d dimensioni, diviene $|x_1|^\alpha + \cdots + |x_d|^\alpha \leq 1$. Per scrivere `lancio`, l'idea è la seguente: si introduce una variabile `somma`, inizialmente nulla, dopodiché un ciclo, si farà carico, per d iterazioni, di:

- generare un numero casuale nell'intervallo $[-1, 1]$;
- aggiornare `somma` aggiungendovi l'α-esima potenza del modulo di tale numero.

Una volta completatasi l'esecuzione del ciclo, basterà guardare se `somma` eccede o meno 1, e restituire `FALSO` o `VERO` di conseguenza.

Il programma avrà infine questo aspetto:

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <iomanip>
#include <cmath>
#include "fiore_random.h"
bool lancio(double alpha, int dimensioni){
   int i; double somma;
   i=0; somma = 0;
   while(i<dimensioni){
      somma = somma + pow(fabs(fiore::drand(-1,1)),alpha);
      i=i+1;
   }
   return somma <= 1;
}
double freq(int N, double alpha, int dimensioni){
   int i, successi;
   i = 0; successi = 0;
   while(i<N){
      if(lancio(alpha,dimensioni)){
         successi = successi+1;
      }
      i = i+1;
   }
   return successi/(double)N;
}
int main(){
   int num_lanci, dimensioni; double alpha;
   double stima_vol, stima_rapp;

   std::cout << std::scientific << std::setprecision(5);

   std::cout << "Numero dei lanci da eseguire: ";
   std::cin >> num_lanci;
   std::cout << "Numero delle dimensioni: ";
   std::cin >> dimensioni;
   std::cout << "Alpha: ";
   std::cin >> alpha;

   stima_rapp = freq(num_lanci,alpha, dimensioni);
   stima_vol = stima_rapp*pow(2,dimensioni);

   std::cout <<"Stima rapp. ipervolumi palla/ipercubo:\t"
             << stima_rapp << "\n";
   std::cout <<"Stima dell'ipervolume della palla:\t"
             << stima_vol << "\n";
   return 0;
}
```

2 Distanze e spazi metrici

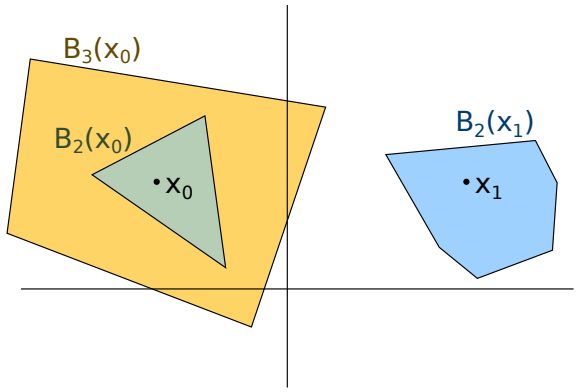
Sia X un insieme. Una *distanza* su X è una funzione $d : X \times X \to [0, +\infty)$ che associa ad ogni coppia (P, Q) di elementi dell'insieme un numero reale positivo o nullo $d(P, Q)$ che chiameremo *distanza* tra P e Q. Per potersi chiamar tale, una distanza deve soddisfare gli assiomi che seguono.

- Simmetria: $d(P, Q) = d(Q, P)\ \forall P, Q \in X$. Se P è la casa e Q è la scuola, questo significa che la distanza tra casa e scuola dev'essere uguale alla distanza tra scuola e casa.
- Diseguaglianza triangolare: $d(P, Q) \leq d(P, R) + d(R, Q)\ \forall P, Q, R \in X$. Se P è la casa, Q è la scuola ed R la pasticceria, significa che andare da casa a scuola passando per la pasticceria richiede un tempo maggiore (o al limite uguale) di quello che occorre per andare da casa a scuola direttamente.
- Non degenerazione: $d(P, Q) = 0 \iff P = Q$. L'implicazione $\Longleftarrow$ è, a ben guardare, una conseguenza dalla diseguaglianza triangolare, e non richiederebbe un assioma specifico. Viceversa, il verso $\Longrightarrow$ contiene un'affermazione non ovvia, e cioè che punti distinti non hanno mai distanza nulla l'uno dall'altro.

Un insieme X provvisto di una qualche distanza d si dice *spazio metrico* e si indica con (X, d). Dato uno spazio metrico (X, d), fissato un punto x_0 (che chiameremo *centro*) e un numero reale $r > 0$ (che chiameremo *raggio*), la *palla chiusa di centro* x_0 *e raggio* r è il luogo dei punti che distano al più r da x_0:

$$\bar{B}_r(x_0) := \{x \in X : d(x, x_0) \leq r\}$$

Si immagini ora di fissare $X = \mathbb{R}^2$, per capire che cosa possa essere una distanza sul piano. Osserviamo, come prima cosa, che le palle di una stessa distanza possono avere le forme più variegate; per esempio, si può costruire una distanza sul piano che ammetta come palle le figure qui sotto disegnate:

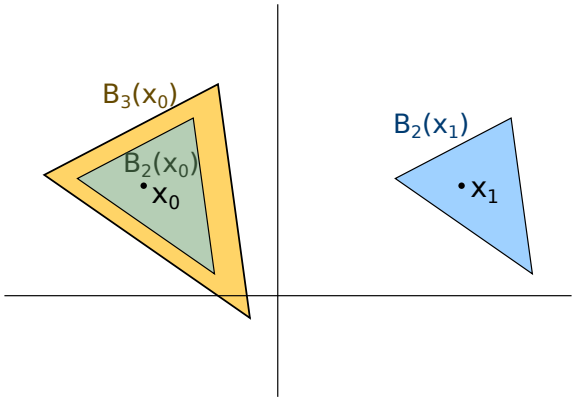


La nozione di distanza appena introdotta si dimostra dunque eccessivamente "selvatica", lontana da ciò che s'intende usualmente con *distanza*. Si pensi, per esempio, alla distanza euclidea: palle di centri e raggi diversi hanno quindi tutte una medesima forma (quella del cerchio), situazione ben differente da quella sopra raffigurata...

Occorre dunque domandarsi che cosa vi sia di manchevole, o meglio di eccessivamente permissivo, nella definizione che si è data di distanza. Il problema si può rintracciare, in vero, già nella premessa "sia X un insieme": la nostra definizione vede il piano $X = \mathbb{R}^2$ *come un insieme*. Trattare il piano come un insieme è certamente possibile, ma è pure estremamente riduttivo: il piano è sì un insieme di punti, ma un insieme di punti "con un'anima", che li "tiene assieme" secondo una determinata *geometria*. Di tale geometria sono espressione le *trasformazioni affini*, e in particolare le *dilatazioni* e le *traslazioni*. Una "buona" distanza sul piano dovrà allora interagire ragionevolmente con tali trasformazioni, secondo i due assiomi ulteriori che seguono.

- Invarianza traslazionale: per ogni coppia di punti $P, Q \in \mathbb{R}^2$, e per ogni vettore $v \in \mathbb{R}^2$, deve valere che $d(P + v, Q + v) = d(P, Q)$. Cioè le traslazioni non debbono modificare le distanze.
- Omogeneità: per ogni $\lambda \in \mathbb{R} \setminus \{0\}$, e per ogni coppia di punti $P, Q \in \mathbb{R}^2$, deve aversi che $d(\lambda P, \lambda Q) = |\lambda| d(P, Q)$. Dilatando per λ, cioè moltiplicando per λ le coordinate di tutti i punti, tutte le distanze devono risultare riscalate del fattore $|\lambda|$.

L'invarianza traslazionale e l'omogeneità assicurano che le palle della metrica abbiano tutte lo stesso "aspetto": due palle di egual raggio e centri distinti saranno semplicemente l'una la traslata dell'altra; una palla di raggio 2 non sarà nient'altro che una palla di raggio 1 raddoppiata:



Per studiare completamente una "buona" distanza basta dunque esaminarne una *singola* palla: si sceglie, convenzionalmente, la *palla di raggio 1 e centro l'origine*, cui ci si riferisce in genere chiamandola semplicemente *palla*, per antonomasia.

Viene a questo punto spontaneo domandarsi quali figure geometriche possano fungere da palla di una qualche "buona" distanza. Un'importante condizione necessaria è data dalla convessità:

Proposizione. *La palla di una qualunque "buona" distanza sul piano (cioè di una distanza omogenea e invariante per traslazione) è convessa.*

Dimostrazione. Siano P e Q due punti qualsiasi della palla unitaria di centro l'origine $\bar{B}_1(O)$. Dobbiamo mostrare che tutto il segmento PQ è contenuto nella palla. Il generico punto del segmento si può scrivere come $(1-t)P + tQ$, per $t \in [0,1]$ (si osservi che per $t = 0$ si ottiene P, per $t = 1$ si ottiene Q, per valori di t intermedi tra 0 e 1 si ottengono i vari punti interni al segmento...).

Dobbiamo mostrare che tale punto appartiene effettivamente a $\bar{B}_1(O)$, cioè che $d((1-t)P + tQ, O) \leq 1$. Ma:

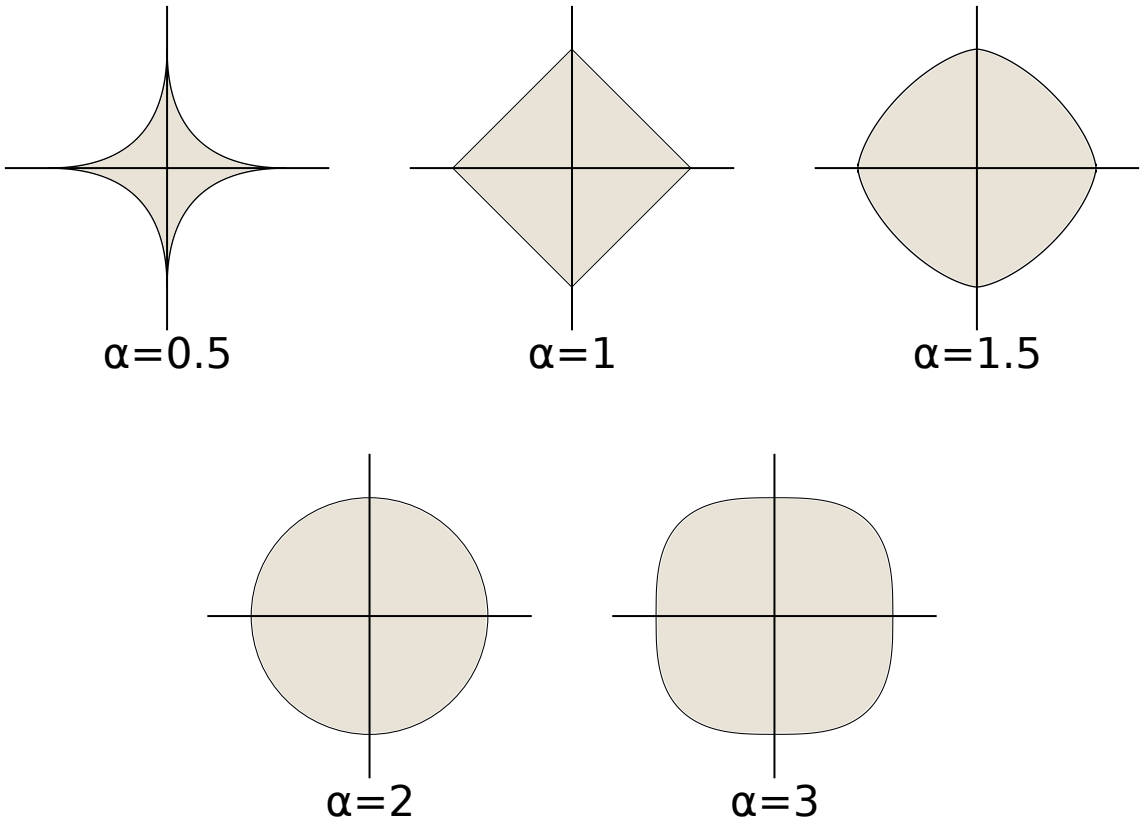
$$\begin{aligned}
d((1-t)P + tQ, O) &= d((1-t)P, -tQ) && \text{per invarianza traslazionale} \\
&\leq d((1-t)P, O) + d(O, -tQ) && \text{per diseguaglianza triangolare} \\
&= (1-t)d(P,O) + td(O,Q) && \text{per omogeneità} \\
&\leq (1-t)\cdot 1 + t \cdot 1 && \text{in quanto } P, Q \in \bar{B}_1(O) \\
&= 1 - t + t = 1
\end{aligned}$$

□

Ottime candidate ad essere esempi di "buone" distanze sul piano sono le d_α presentate nell'esercizio, e così definite:

$$d_\alpha(P,Q) := (|x_P - x_Q|^\alpha + |y_P - y_Q|^\alpha)^{1/\alpha} \qquad \alpha \in \mathbb{R}^+$$

Le loro palle, definite dall'equazione $d_\alpha((x,y),(0,0)) \leq 1$, hanno questo aspetto:

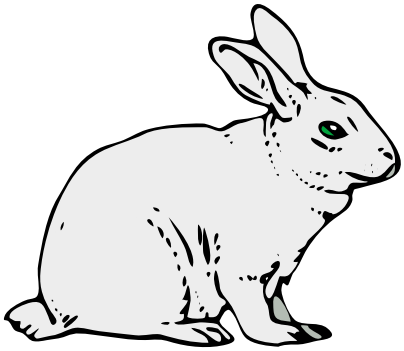


Per $\alpha \geq 1$, d_α è effettivamente una "buona" distanza. Per $\alpha < 1$, invece, questo non può accadere, per via della concavità di quella che dovrebbe essere la palla

unitaria. Più particolarmente, si può verificare che, dei cinque assiomi che abbiamo in precedenza elencato, quattro sono rispettati, ma uno fallisce: in particolare, a non valere nel caso $\alpha < 1$ è la diseguaglianza triangolare.

7 Sistemi dinamici in tempo discreto

1 Conigli



La rigogliosa prateria di Thunder-ten-tronckh ospita una prolifica comunità di conigli. La dimensione della popolazione sarà espressa mediante un numero reale $x \in [0, 1]$: $x = 0$ significa l'estinzione; $x = 1$ significa che i conigli, stipati l'uno contro l'altro, hanno esaurito tutto lo spazio a loro disposizione.

Il professor Pangloss, dopo anni di studio, si è convinto di aver individuato la legge d'evoluzione cui il numero di conigli ubbidisce. Potremmo enunciarla come segue: se $x \in [0, 1]$ è il numero di conigli nell'anno t, l'anno successivo (cioè l'anno $t + 1$) saranno $2.5 \cdot x(1 - x)$.

1. Si scriva una funzione `f` che, accettata come parametro la popolazione x in un certo anno, restituisca al chiamante la popolazione nell'anno successivo.

2. Si scriva un `main` che, chiesta all'utente la numerosità della popolazione nell'anno $t = 0$, tracci un grafico del numero di conigli in funzione del tempo, per gli anni da $t = 0$ a $t = 50$. Per disegnare un punto del grafico, si usi la funzione `fiore::disegna`, che accetta come parametri l'ascissa e l'ordinata del punto[(a)]. Per visualizzare il grafico, bisogna aggiungere, al termine del `main`, le istruzioni:

```
fiore::g("set size ratio 0.5"); //Per la forma dell'area di disegno
fiore::g("set yrange [0:1]"); //Per impostare gli estremi dell'asse y
fiore::grafico("with linespoints"); //Per disegnare con linee e punti
```

   Si provi ad eseguire il programma più volte, variando la popolazione iniziale. Cosa si osserva? Qual è, di volta in volta, il "destino" della popolazione sul lungo termine?

3. Pangloss ritratta in parte la propria conclusione; in particolare, non è più sicuro che il giusto coefficiente da inserire nella legge di evoluzione (che denoteremo d'ora innanzi con λ) sia proprio 2.5. Si generalizzi debitamente il programma scritto, di modo che consenta a Pangloss di indicare a terminale non solo la popolazione iniziale, ma anche il valore di λ.

   Si esegua il programma immettendo come popolazione iniziale 0.2, 0.5 e 0.7, e per λ i valori 0.5, 1.5, 2.5, 3.3, 3.5, 3.68, 3.75, 3.83, 3.9. **Si dica, per ogni valore di λ, cosa si osserva**.

   Si osservi che i soli valori che ha senso fissare per λ sono quelli dell'intervallo reale $[0, 4]$. Perché?

[(a)] Le funzioni grafiche cui si fa riferimento appartengono alla libreria `fiore_gnuplot.h`, che si può scaricare dal sito del corso (`https://sites.google.com/a/voltaweb.it/amat1718cpp`). Quando la libreria `fiore_gnuplot.h` è inclusa, occorre compilare usando la combinazione di tasti `Shift+F9`, in luogo del semplice `F9`.

4. Pangloss, disorientato, vuol vederci più chiaro: fissa allora il valore della popolazione iniziale a 0.5, e desidera un grafico che descriva sinteticamente il destino della popolazione in funzione di λ.

   Per ogni valore del parametro $\lambda \in [0, 4]$, che riporteremo in ascissa, vengono allora disegnati i punti $(\lambda, x_{100}), (\lambda, x_{101}), \ldots, (\lambda, x_{150})$, ove le ordinate corrispondono alla numerosità della popolazione negli anni $t = 100, \ldots, t = 150$. Per tracciare il grafico, si usi come passo $\Delta\lambda = 0.001$.

   Si suggerisce di utilizzare, al termine del `main`, i seguenti comandi di formattazione e visualizzazione del grafico:

```
fiore::g("set size ratio 0.5"); //Per la forma dell'area di disegno
fiore::g("set yrange [0:1]"); //Per impostare gli estremi dell'asse y
fiore::grafico("with points pt 0"); //Per disegnare con soli punti
```

   Cosa si osserva?

   Si arricchisca il programma in modo che faccia scegliere all'utente l'intervallo di valori di λ da rappresentare, mantenendo però sempre un passo pari alla quattromillesima parte dell'intervallo.