Tutorato di Informatica

Foglio 1 - Rappresentazione dell'Informazione e Aritmetica Binaria

Matematici I Anno

4 Marzo 2026

"Ci sono 10 tipi di persone al mondo, chi conosce i numeri primi e chi no."

Esercizio 00 - Riscaldamento

Testo: Quante potenze di 2 riesci a ricordare a memoria? Scoprilo adesso :)

PARTE 1: Rappresentazione Numeri Interi Positivi e Testo

Esercizio 01

Testo: Effettuare la conversione indicata dal binario puro al decimale: $10010010_2 =?_{10}$ (sol 146)

Spiegazione: Espansione polinomiale in base 2: $1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 128 + 16 + 2 = \mathbf{146}$.

Esercizio 02

Testo: Indicare quali sono il più grande ed il più piccolo valore (in base 10) che si possono rappresentare in ciascuna delle seguenti codifiche: binaria a 16 bit senza segno; binaria a 16 bit con complemento a 2.

Spiegazione:

- *Unsigned:* $n = 16$ bit coprono l'intervallo $[0, 2^{16} - 1]$. Risultato: **0** e **65535**.
- *Complemento a 2:* L'intervallo è $[-2^{n-1}, 2^{n-1} - 1]$. Risultato: $-$**32768** e **32767**.

Esercizio 03

Testo: Effettuare la conversione indicata: $F3B4C_{16} =?_2$ (sol 1111 0011 1011 0100 1100)

Spiegazione: Ogni cifra esadecimale corrisponde esattamente a un "nibble" (4 bit) poiché $16 = 2^4$. $F \to 1111, 3 \to 0011, B \to 1011, 4 \to 0100, C \to 1100$.

Esercizio 04

Testo: Codificare in numeri esadecimali e decimali secondo lo standard ASCII la stringa: `x=7*(y)`

Spiegazione:

Carattere	x	=	7	*	(	y	)
Hex	78	3D	37	2A	28	79	29
Dec	120	61	55	42	40	121	41

PARTE 2: Rappresentazione Numeri Interi Positivi/Negativi

Esercizio 05

Testo: Convertire i seguenti numeri in base 10 nella rappresentazione Modulo e Segno (MS) a 8 bit: 59 e -20.

Spiegazione: In MS, il bit più significativo (MSB) è il segno (0 per +, 1 per -).

- $59_{10} = 32 + 16 + 8 + 2 + 1 = 00111011_2$. Segno + (0) $\rightarrow$ **00111011**.
- -20_{10}: $20 = 16 + 4 = 00010100_2$. Segno − (1) $\rightarrow$ **10010100**.

Esercizio 06

Testo: Convertire in CA2 a 8 bit: 33_{10}, $-\langle 14 \rangle_{16}$ e -20_{10}.

Spiegazione:

- 33_{10}: Poiché è positivo, coincide con il binario puro $\rightarrow$ **00100001**.
- -14_{16}: Corrisponde a -20_{10}. Procedura: 20_{10} = 00010100. Inverti bit: 11101011. Aggiungi 1: **11101100**.
- -20_{10}: Stesso calcolo precedente $\rightarrow$ **11101100**.

Esercizio 07

Testo: Per ciascuna coppia di numeri binari a 5 bit, indicare quale dei due è più grande nel caso in cui li si consideri codificati con segno, e poi codificati con complemento a 2 (a 5 bit): 00101 vs 11101 e 10110 vs 11010.

Spiegazione:

- **Coppia 1:** 00101(+5) vs 11101. In MS 11101 = −13. In CA2 11101 = −3. In entrambi i casi +5 è maggiore.
- **Coppia 2:** 10110 vs 11010. In MS: −6 vs −10 $\rightarrow$ maggiore −6. In CA2: −10 (10110) vs −6 (11010) $\rightarrow$ maggiore −6 (11010).

PARTE 3: Numeri Frazionari

Esercizio 08

Testo: Effettuare le seguenti conversioni tra decimali e binari a virgola fissa: $10.01_2 \rightarrow ?_{10}$ e $10.125_{10} \rightarrow ?_2$.

Spiegazione:

- $10.01_2 = 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 2 + 0.25 = \mathbf{2.25_{10}}$.
- 10.125_{10}: $10 = 1010_2$. $0.125 = 1/8 = 2^{-3} = 0.001_2$. Risultato: $\mathbf{1010.001_2}$.

Esercizio 09: Decodifica IEEE-754

Esercizio: Interpretazione IEEE-754 Single Precision

Testo:
Interpretare il seguente numero esadecimale come numero a virgola mobile in standard IEEE 754 a precisione singola (32 bit: 1 segno, 8 esponente con bias 127, 23 mantissa). Mostrare i passaggi per:

1. Estrarre i bit di segno, esponente e mantissa;
2. Calcolare l'esponente reale;
3. Scrivere il numero in notazione scientifica binaria;
4. Convertire il numero in decimale.

$0x41340000$

Svolgimento Dettagliato:

1. **Passaggio 1: Conversione da Esadecimale a Binario**
   Ogni cifra esadecimale corrisponde esattamente a 4 bit. Convertiamo sistematicamente:
   - $4_{16} = 0100_2$
   - $1_{16} = 0001_2$
   - $3_{16} = 0011_2$
   - $4_{16} = 0100_2$
   - $0_{16} = 0000_2$ (ripetuto per le restanti 4 posizioni)

   Otteniamo la stringa completa a 32 bit:

   `0100 0001 0011 0100 0000 0000 0000 0000`

2. **Passaggio 2: Suddivisione dei Campi IEEE-754**
   Lo standard suddivide i 32 bit in tre parti: Segno (1 bit), Esponente (8 bit) e Mantissa (23 bit).

   $$\underbrace{0}_{S} \mid \underbrace{10000010}_{E} \mid \underbrace{01101000000000000000000}_{M}$$

3. **Passaggio 3: Analisi del Segno (S)**
   Il bit di segno è $S = 0$. Nello standard IEEE-754, 0 indica un numero **positivo**.

4. **Passaggio 4: Calcolo dell'Esponente Reale**
   L'esponente memorizzato (E) è in eccesso (bias) di 127.

- Valore binario di E: 10000010_2
- Conversione decimale: $1 \cdot 2^7 + 1 \cdot 2^1 = 128 + 2 = 130_{10}$
- Sottrazione del Bias: $exp_{reale} = E - 127 = 130 - 127 = 3$

5. **Passaggio 5: Ricostruzione della Mantissa (con Hidden Bit)**
   La mantissa normalizzata prevede sempre un bit 1 implicito prima della virgola (non memorizzato). Prendiamo i primi bit significativi di M (01101):

$$\text{Valore normalizzato} = 1.M = 1.01101_2$$

6. **Passaggio 6: Calcolo del Valore in Binario "Puro"**
   Uniamo segno, mantissa ed esponente reale:

$$V = +1.01101_2 \times 2^3$$

   Moltiplicare per 2^3 equivale a spostare la virgola di 3 posizioni verso destra:

$$1.01101 \xrightarrow{\text{shift } 3} 1011.01_2$$

7. **Passaggio 7: Conversione Finale in Decimale**
   Convertiamo separatamente la parte intera e la parte frazionaria:

   - **Parte intera (1011_2):** $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 0 + 2 + 1 = 11_{10}$
   - **Parte frazionaria (0.01_2):** $0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 0 + \frac{1}{4} = 0.25_{10}$

   Il valore decimale finale è: **11.25**.

Esercizio 10 - Conversione in Virgola Fissa (Metodo Algoritmico)

Testo:
Qual è la rappresentazione in virgola fissa (con 8 bit per la parte intera e 8 bit per la parte decimale) del numero 23.625_{10}?

Svolgimento:

1. **Conversione della Parte Intera (23) - Divisioni Successive:**
   Dividiamo ripetutamente per 2 e raccogliamo i resti dal basso verso l'alto:

$$\begin{aligned} 23 \div 2 &= 11 \quad \text{resto } \mathbf{1} \uparrow \text{ (LSB)} \\ 11 \div 2 &= 5 \quad \text{resto } \mathbf{1} \\ 5 \div 2 &= 2 \quad \text{resto } \mathbf{1} \\ 2 \div 2 &= 1 \quad \text{resto } \mathbf{0} \\ 1 \div 2 &= 0 \quad \text{resto } \mathbf{1} \text{ (MSB)} \end{aligned}$$

   Parte intera ottenuta: 10111_2.
   Estesa a 8 bit (padding a sinistra): **00010111**.

2. **Conversione della Parte Frazionaria (0.625) - Moltiplicazioni Successive:**
   Moltiplichiamo la parte frazionaria per 2. La parte intera del risultato diventa il bit binario, la parte frazionaria rimanente viene usata per il passaggio successivo:

$$\begin{aligned} 0.625 \times 2 &= \mathbf{1}.25 \quad \rightarrow \text{bit } \mathbf{1} \downarrow \\ 0.25 \times 2 &= \mathbf{0}.5 \quad \rightarrow \text{bit } \mathbf{0} \\ 0.5 \times 2 &= \mathbf{1}.0 \quad \rightarrow \text{bit } \mathbf{1} \end{aligned}$$

   La parte frazionaria è diventata zero, il processo termina.
   Parte frazionaria ottenuta: $.101_2$.
   Estesa a 8 bit (padding a destra): **10100000**.

3. **Composizione del Risultato:**
   Unendo le due parti separate dal punto binario, otteniamo la stringa finale:

$$\mathbf{00010111.10100000_2}$$

Verifica Decimale:

$$(00010111) = 16 + 4 + 2 + 1 = 23$$

$$(0.101) = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.5 + 0 + 0.125 = 0.625$$

Esercizio 11

Testo: Convertire il numero binario 0 10000001 11110000000000000000000 in virgola mobile (floating point, single precision) codificato con lo standard IEEE-754 in decimale.

Spiegazione: $S = 0(+)$. $E = 129 - 127 = 2$. $M = 1.1111$. In notazione scientifica: $1.1111_2 \times 2^2$. Spostando la virgola: $111.11_2 = 4 + 2 + 1 + 0.5 + 0.25 = \mathbf{7.75}$.

PARTE 4: Operazioni Binarie e Logica Hardware

Esercizio 12 - La Fisica dell'Overflow

Testo: In quali casi bisogna verificare che non ci sia overflow? (Somma/sottrazione di concordi/discordi).

Spiegazione Standard: L'overflow avviene quando il risultato esce dall'intervallo rappresentabile.

- Somma di concordi (Pos+Pos o Neg+Neg): **Rischio Overflow**.
- Somma di discordi (Pos+Neg): **Mai Overflow**.
- Sottrazione: $A - B$ equivale a $A + (-B)$. Quindi sottrarre un discorde equivale a sommare un concorde $\rightarrow$ **Rischio Overflow**.

Esercizio 13

Testo: Calcolare il risultato della somma tra numeri positivi a 8 bit, indicando il risultato in esadecimale: $10010001_2 + 01001101_2 = ?_{16}$

Spiegazione: Somma bit a bit: 11011110_2. Convertendo in Hex: $1101 = D, 1110 = E \rightarrow$ **0xDE**.

Esercizio 14

Testo: Calcolare $0xA789 + 0x0987 = ?_{16}$. Partendo dalle cifre più a destra, considerando i riporti, e che le cifre esadecimali sono 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Spiegazione: Somma in colonna Hex: $9+7 = 16$ (scrivo 0, riporto 1). $8+8+1 = 17$ (scrivo 1, riporto 1). $7 + 9 + 1 = 17$ (scrivo 1, riporto 1). $A + 0 + 1 = B$. Risultato: **0xB110**.

Esercizio 15

Testo: $0xAB - 0x23 = ?_{16}$

Spiegazione: $B - 3 = 8$. $A - 2 = 8$. Risultato: **0x88**.

Esercizio 16

Testo: Calcolare il risultato della seguente sottrazione tra numeri binari naturali (unsigned), indicando il risultato in CA2 in 8 bit: $1000011_2 - 10011010_2 = ?_2$

Spiegazione:
Poiché il sottraendo (154_{10}) è maggiore del minuendo (67_{10}), l'operazione causerebbe un *underflow* in binario puro. Utilizziamo la rappresentazione in Complemento a 2 (CA2) su 8 bit per gestire il risultato negativo:

$$A - B = A + \text{CA2}(B)$$

1. **Conversione del sottraendo in CA2:** Prendiamo $B = 10011010_2$ (154_{10}).
   - Invertiamo i bit (CA1): 01100101_2
   - Sommiamo 1: $01100101_2 + 1 = 01100110_2$

   Quindi, $\text{CA2}(10011010_2) = 01100110_2$.

2. **Esecuzione della somma (su 8 bit):**

$$\begin{array}{rl} 01000011_2 & (67_{10}) \\ +01100110_2 & (-154_{10} \text{ espresso in CA2}) \\ \hline 10101001_2 & (-87_{10}) \end{array}$$

Verifica:

- $1000011_2 = 64 + 2 + 1 = 67_{10}$
- $10011010_2 = 128 + 16 + 8 + 2 = 154_{10}$
- Differenza: $67 - 154 = -87_{10}$
- Conversione Risultato (10101001_2 in CA2): $-128 + 32 + 8 + 1 = -87_{10}$.

Approfondimento: I 3 Metodi per la Sottrazione Binaria

Per comprendere a fondo la sottrazione binaria, calcoliamo $53_{10} - 35_{10} = 18_{10}$ operando su registri a 8 bit. I due numeri in binario sono:

- Minuendo (A): $53_{10} = 00110101_2$
- Sottraendo (B): $35_{10} = 00100011_2$

Metodo 1: Sottrazione in Colonna con il "Trucco del 2" Invece di ragionare sui prestiti usando il valore binario 10_2, si utilizza un trucco didattico: ogni volta che si chiede un prestito alla colonna di sinistra, si "cancella" lo 0 e si scrive esplicitamente un **2** decimale sopra la colonna corrente. Questo rende il calcolo mentale identico a quello delle elementari.

$$
\begin{array}{cccccccccc}
 & & & & & & 0 & 2 & & \\
 & 0 & 0 & 1 & 1 & 0 & \not{1} & \not{0} & 1 \\
- & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
\hline
= & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0}
\end{array}
$$

Spiegazione passaggio critico: Nella seconda colonna da destra, dobbiamo fare $0 - 1$. Chiediamo in prestito 1 al vicino di sinistra (il bit 1, che viene cancellato e diventa 0). Il prestito vale una "base", cioè 2. Lo scriviamo in rosso sopra lo zero. Facciamo quindi mentalmente $2 - 1 = 1$ e lo scriviamo nel risultato.

Metodo 2: Il Metodo Matematico (Definizione di Complemento a 2^n) Per gli studenti di matematica, è fondamentale capire che non esistono "trucchi magici". La sottrazione $A - B$ in un calcolatore a n bit sfrutta l'aritmetica modulare nell'anello $\mathbb{Z}/2^n\mathbb{Z}$. Invece di sottrarre B, si aggiunge il suo complemento rispetto alla base, definito rigorosamente come $2^n - B$.

Calcoliamo $A - B$ a 8 bit ($n = 8$) usando la formula: $\mathbf{A - B \equiv A + (2^8 - B)} \pmod{\mathbf{2^8}}$

1. **Calcolo dell'inverso additivo:**

   $$2^8 - B = 256 - 35 = 221_{10}$$

   Convertiamo questo valore in binario: $221_{10} = 11011101_2$.

2. **Esecuzione della somma:** Sommiamo A con il valore appena trovato:

   $$53 + 221 = 274_{10} \implies 00110101_2 + 11011101_2 = \mathbf{1}00010010_2$$

3. **Applicazione del modulo:** L'operazione modulo 2^8 equivale fisicamente a "tagliare" il 9° bit (il riporto che eccede il registro). $274 \pmod{256} = 18_{10}$. In binario, scartando l'1 iniziale, otteniamo esattamente $\mathbf{00010010_2}$.

Metodo 3: L'Algoritmo Hardware (NOT + 1) Il Metodo 2 ci dimostra *perché* funziona la matematica, ma la CPU non può calcolare $256 - 35$, altrimenti avrebbe bisogno di un circuito di sottrazione! L'ingegneria informatica usa quindi un teorema: $2^n - B$ è sempre uguale a $\text{NOT}(B) + 1$. Questo è il calcolo effettivo eseguito dalla ALU (Arithmetic Logic Unit).

1. **Inversione logica (NOT):** Invertiamo i bit di B.
   $B = 00100011 \implies \text{NOT}(B) = 11011100_2$

2. **Somma di 1:**
   $11011100 + 1 = 11011101_2$ (Si noti che questo è esattamente il 221_{10} calcolato al Metodo 2!).

3. **Somma finale scartando il riporto:**
   $00110101_2 + 11011101_2 = \mathbf{1} \mid \mathbf{00010010_2}$.

Esercizio 17

Testo: Eseguire $53_{10} - 35_{10}$ in CA2 a 8 bit.

Spiegazione: $53 = 00110101$. -35: $35 = 00100011 \rightarrow \text{NOT}+1 \rightarrow 11011101$. Somma: $00110101 + 11011101 = (1)00010010$. Ignora il riporto. Risultato: **18**.

Esercizi Extra Per i Più Coraggiosi

01. La Morte dell'Associatività: Quando l'Algebra Fallisce

Testo: Nei vostri corsi di Algebra state studiando le proprietà dei campi ($\mathbb{R}$, $\mathbb{Q}$, $\mathbb{C}$). Uno degli assiomi intoccabili è la proprietà associativa dell'addizione: $(A + B) + C = A + (B + C)$.

Eppure, nel mondo dell'informatica, l'insieme dei numeri floating-point **non forma un campo** e questa uguaglianza non vale! Il processore, per colpa della memoria limitata, commette dei clamorosi errori di "assorbimento" quando somma numeri di grandezze molto diverse.

Riuscite a distruggere la proprietà associativa? Trovate tre numeri reali A, B e C tali per cui il calcolatore vi darà due risultati completamente diversi a seconda di come mettete le parentesi.

Il Concetto Matematico (L'Allineamento degli Esponenti): Per sommare due numeri in notazione scientifica binaria della forma $m_1 \times 2^{e_1}$ e $m_2 \times 2^{e_2}$ (supponiamo $e_1 > e_2$), la CPU deve prima "allinearli" in modo che abbiano lo stesso esponente. Per farlo, "shifta" (fa scorrere) la mantissa del numero più piccolo verso destra di $e_1 - e_2$ posizioni. Poiché in precisione singola la mantissa ha solo 24 bit di spazio fisico, se la differenza tra gli esponenti è troppo grande, i bit del numero piccolo scivolano letteralmente fuori dal registro e finiscono nel nulla. Il numero piccolo viene "assorbito" e approssimato a zero.

La Dimostrazione (Il Controesempio): Seguiamo il suggerimento e poniamo $A = 10^{20}$, $B = -10^{20}$ e $C = 1$.

Analizziamo il lato sinistro dell'uguaglianza, $(A + B) + C$:

1. Sommiamo prima l'oceano e il suo opposto: $10^{20} + (-10^{20}) = 0$. I due numeri hanno la stessa identica magnitudine, quindi si annullano perfettamente senza perdite di precisione.
2. Aggiungiamo la goccia: $0 + 1 = \mathbf{1}$.

Analizziamo ora il lato destro, $A + (B + C)$:

1. Sommiamo l'oceano negativo e la goccia: $-10^{20}+1$. Per allineare l'1 all'ordine di grandezza di 10^{20} servirebbero circa 67 bit di mantissa (poiché $10^{20} \approx 2^{66}$). Avendone a disposizione solo 24, l'1 scorre fuori dal registro e viene inesorabilmente cancellato. Il computer calcola spietatamente $-10^{20} + 1 \approx -10^{20}$.
2. Aggiungiamo l'oceano positivo: $10^{20} + (-10^{20}) = \mathbf{0}$.

Poiché $1 \neq 0$, abbiamo appena dimostrato che la proprietà associativa fallisce. In informatica, l'ordine in cui sommate una serie di dati può farvi perdere completamente le informazioni più piccole!

02. Densità Asimmetrica (La Topologia del Floating Point)

Domanda: Consideriamo la densità dei numeri IEEE-754 (singola precisione) sulla retta reale. Ci sono più numeri rappresentabili nell'intervallo $[0, 1)$ o nell'intervallo $[1, 2)$?

Il Concetto Matematico (Distribuzione Logaritmica): A differenza di $\mathbb{R}$, dove tra due numeri distinti ci sono infiniti elementi, o dei numeri a virgola fissa che hanno una spaziatura uniforme, lo standard IEEE-754 distribuisce i numeri in modo logaritmico. La densità dei numeri esplode man mano che ci si avvicina allo zero. Ogni numero è codificato come $\pm 1.M \times 2^E$. La mantissa M è fissa a 23 bit. Questo implica un teorema fondamentale: *per ogni valore fissato dell'esponente E, l'intervallo coperto contiene esattamente 2^{23} numeri equidistanti.*

La Dimostrazione: Confrontiamo i due intervalli richiesti:

- **Intervallo $[1, 2)$:** Per mappare un numero in questo intervallo, l'esponente E deve essere obbligatoriamente 0. Infatti, i numeri generati sono della forma $1.M \times 2^0$, che spaziano da $1.000...0_2$ a $1.111...1_2$. Essendoci un solo esponente possibile ($E = 0$), ci sono esattamente $\mathbf{1 \times 2^{23}}$ numeri rappresentabili.

- **Intervallo** $[0, 1)$**:** Per cadere in questo intervallo, l'esponente E deve essere strettamente negativo.
  - Se $E = -1$, otteniamo $1.M \times 2^{-1}$, coprendo l'intervallo $[0.5, 1)$. (Ci sono 2^{23} numeri qui).
  - Se $E = -2$, otteniamo $1.M \times 2^{-2}$, coprendo l'intervallo $[0.25, 0.5)$. (Altri 2^{23} numeri).
  - Se $E = -3$, copriamo $[0.125, 0.25)$... e così via.

  Lo standard permette esponenti negativi fino a $E = -126$. Pertanto, ci sono ben 126 intervalli (o "bin") di esponenti diversi che mappano valori in $[0, 1)$.

Conclusione: L'intervallo $[0, 1)$ contiene $\mathbf{126 \times 2^{23}}$ numeri normalizzati (più i denormalizzati prossimi allo zero), risultando oltre 126 volte più "denso" dell'intervallo $[1, 2)$.

03. Il Criterio di Divisibilità Binario (Aritmetica Modulare)

Testo: Sicuramente ricorderai un trucco famoso in base 10: un numero è divisibile per 11 se la somma a segni alterni delle sue cifre fa 0 (o un multiplo di 11).

Proviamo a fare lo stesso gioco in base 2: se prendiamo un numero binario ed eseguiamo la somma a segni alterni dei suoi bit, stiamo in realtà applicando un test per verificare la divisibilità per un "divisore misterioso". Qual è questo divisore? Fai qualche tentativo, trova il divisore e dimostra che la regola funziona sempre, sfruttando la scrittura polinomiale dei numeri in base 2 e le proprietà dell'aritmetica modulare.

La Risposta: Il "divisore misterioso" è il **3**.

La Dimostrazione (Congruenze e Anelli): Sia N un intero positivo. Sviluppiamo N in base 2 come un polinomio valutato in $x = 2$:

$$N = \sum_{i=0}^{k} b_i 2^i = b_0 2^0 + b_1 2^1 + b_2 2^2 + b_3 2^3 + \dots$$

dove $b_i \in \{0, 1\}$.

Vogliamo studiare le proprietà di N nell'anello delle classi di resto modulo 3 ($\mathbb{Z}/3\mathbb{Z}$). Applicando l'operatore modulo all'equazione, possiamo usare le proprietà degli omomorfismi per portare il modulo all'interno della sommatoria. Sappiamo che in $\mathbb{Z}/3\mathbb{Z}$, la classe di equivalenza di 2 è congrua a -1:

$$2 \equiv -1 \pmod 3$$

Sostituendo 2 con -1 nel polinomio, otteniamo:

$$N \equiv \sum_{i=0}^{k} b_i (-1)^i \pmod 3$$

Sviluppando i termini, le potenze dispari di -1 restano negative, quelle pari diventano positive:

$$N \equiv b_0(1) + b_1(-1) + b_2(1) + b_3(-1) + \dots \pmod 3$$

$$N \equiv b_0 - b_1 + b_2 - b_3 + \dots \pmod 3$$

Abbiamo dimostrato che N e la somma a segni alterni dei suoi bit appartengono alla stessa classe di congruenza modulo 3. Se la somma alternata è 0 (o un multiplo di 3), allora $N \equiv 0 \pmod 3$.

Verifica Numerica: Sia $N = 9_{10}$, che in binario si scrive 1001_2. Partendo dal bit meno significativo (a destra): $1 - 0 + 0 - 1 = 0$. Essendo $0 \equiv 0 \pmod 3$, il numero di partenza è divisibile per 3.

04. Numeri senza Segno: La Magia della Base Negabinaria

Testo: Siamo abituati a usare basi positive (come 10 o 2). Di conseguenza, se vogliamo scrivere un numero negativo, siamo costretti ad aggiungere un segno "$-$", oppure a inventarci dei "trucchi" informatici complessi come il Complemento a 2 per incastrare tutto in memoria.

Ma cosa succede se usiamo una base negativa? Consideriamo un sistema posizionale in base -2. Le cifre a nostra disposizione sono sempre e solo 0 e 1, ma i pesi delle varie posizioni sono potenze di (-2). Qual è il grande vantaggio matematico di questa insolita struttura? Per intuirlo e prendere confidenza con questo sistema "alieno", accetta la sfida: prova a scrivere il numero decimale 10 in base -2.

Suggerimento: scrivi le prime potenze di (-2) *(ricordando che le potenze pari saranno positive e quelle dispari negative) e cerca la combinazione giusta di* 0 *e* 1 *per raggiungere esattamente il* 10*.*

Il Vantaggio Teorico: A differenza delle basi positive che richiedono un bit aggiuntivo per il segno o un'aritmetica modulare complessa (come il complemento a 2) per mappare $\mathbb{Z}$ su un numero finito di bit, la base negabinaria permette di rappresentare **qualsiasi numero intero** $z \in \mathbb{Z}$ (positivo o negativo) in modo univoco e naturale, usando solo 0 e 1, senza alcun "artificio" per i segni.

Costruzione dell'Algoritmo: I pesi delle posizioni in base -2 si calcolano valutando $(-2)^i$:

- $i = 0$: $(-2)^0 = 1$
- $i = 1$: $(-2)^1 = -2$
- $i = 2$: $(-2)^2 = 4$
- $i = 3$: $(-2)^3 = -8$
- $i = 4$: $(-2)^4 = 16$

Si nota immediatamente che i bit di indice pari forniscono addendi positivi, mentre i bit di indice dispari forniscono addendi negativi.

Cerchiamo la combinazione lineare con coefficienti in $\{0, 1\}$ la cui somma dia 10:

1. Abbiamo bisogno di un valore positivo grande. Accendiamo il bit $i = 4$ (peso 16). Somma parziale: 16. (Siamo in eccesso di 6).
2. Dobbiamo sottrarre. Accendiamo il bit $i = 3$ (peso -8). Somma parziale: $16 - 8 = 8$. (Siamo in difetto di 2).
3. Dobbiamo aggiungere. Accendiamo il bit $i = 2$ (peso 4). Somma parziale: $8 + 4 = 12$. (Siamo in eccesso di 2).
4. Dobbiamo sottrarre. Accendiamo il bit $i = 1$ (peso -2). Somma parziale: $12 - 2 = \mathbf{10}$. (Target raggiunto).
5. Il bit $i = 0$ (peso 1) non ci serve e rimane spento (0).

Leggendo i coefficienti da sinistra (indice massimo) a destra (indice minimo), la rappresentazione di 10_{10} in base -2 è **11110**.

05. Il Segreto del Complemento a 2: L'Algebra dietro l'Hardware

Testo: Negli esercizi precedenti vi è stata data una "regola magica" calata dall'alto: per trovare l'opposto di un numero binario (ovvero cambiargli il segno), basta invertire tutti i bit (operazione NOT) e aggiungere 1. In matematica, però, non si accetta nulla per fede.

Sappiamo che un registro hardware a n bit non è altro che una perfetta rappresentazione fisica dell'anello delle classi di resto modulo 2^n ($\mathbb{Z}/2^n\mathbb{Z}$). La sfida è questa: dimostrate rigorosamente che l'algoritmo informatico "$\text{NOT}(x) + 1$" equivale esattamente a calcolare l'inverso additivo di x in quell'anello algebrico.

Suggerimento: prendete un numero x e il suo inverso logico $NOT(x)$. Se li sommate in colonna bit a bit, che numero ottenete sempre? Partite da questa equazione...

La Dimostrazione: Sia x un intero codificato su n bit: $x = b_{n-1}b_{n-2}\dots b_0$. Il nostro ambiente di lavoro è un registro hardware di n bit, che modella matematicamente l'anello $\mathbb{Z}/2^n\mathbb{Z}$. In questa struttura, i valori che superano o uguagliano 2^n provocano un *overflow* e "ripartono da zero".

Iniziamo seguendo il suggerimento e analizziamo l'operazione bit a bit $\text{NOT}(x)$. Questa operazione sostituisce ogni 1 con 0 e ogni 0 con 1. Se calcoliamo la somma algebrica $x+\text{NOT}(x)$, ogni singola colonna della somma conterrà esattamente uno 0 e un 1. Pertanto, la somma di ogni colonna farà 1 senza mai generare alcun riporto.

Il risultato sarà una stringa composta da esattamente n cifre uguali a 1 (ad esempio, se $n = 4$: $1010_2 + 0101_2 = 1111_2$). Sappiamo che il valore numerico di una stringa di n uni è il valore massimo rappresentabile in binario puro, ovvero $2^n - 1$. Abbiamo appena scoperto il nostro lemma fondamentale:

$$x + \text{NOT}(x) = 2^n - 1$$

Aggiungiamo ora 1 a entrambi i membri dell'equazione:

$$x + \text{NOT}(x) + 1 = 2^n$$

Isoliamo a sinistra i termini che rappresentano l'algoritmo hardware, ovvero $\text{NOT}(x) + 1$, spostando x a destra:

$$\text{NOT}(x) + 1 = 2^n - x$$

A questo punto entra in gioco l'algebra. Poiché l'hardware opera rigorosamente in $\mathbb{Z}/2^n\mathbb{Z}$, applichiamo l'operatore modulo 2^n a entrambi i membri. Sapendo che nell'anello vale l'equivalenza $2^n \equiv 0 \pmod{2^n}$, l'equazione diventa:

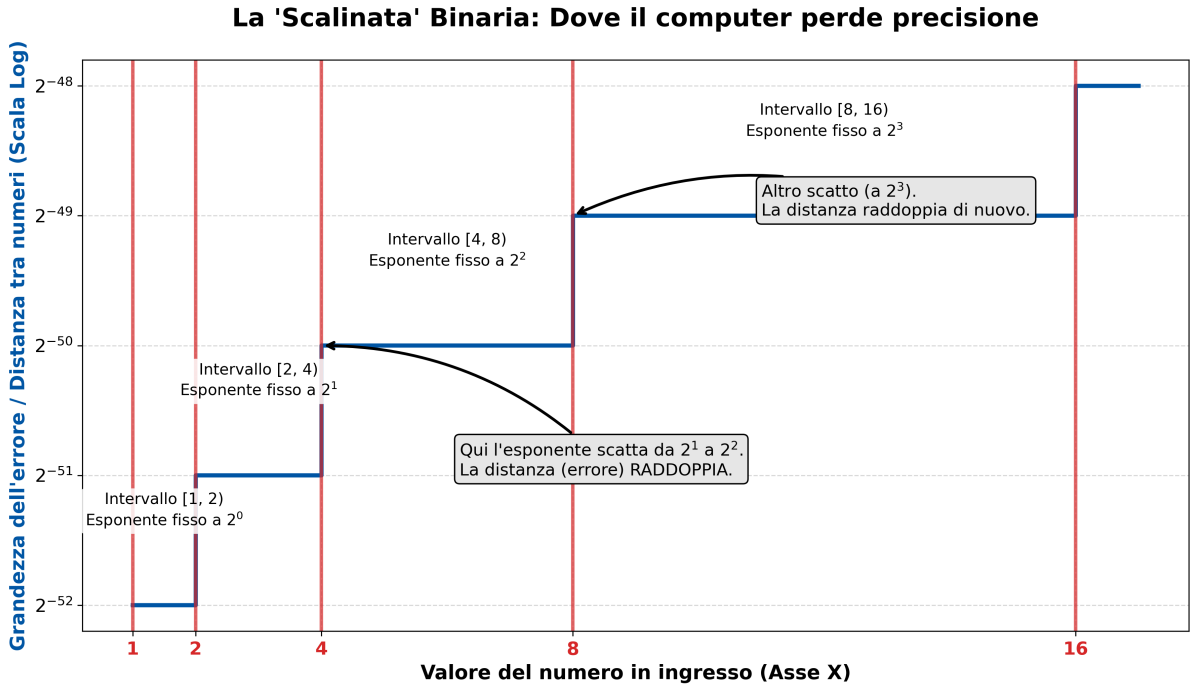
$$\text{NOT}(x) + 1 \equiv 0 - x \pmod{2^n}$$

$$\text{NOT}(x) + 1 \equiv -x \pmod{2^n} \quad \blacksquare$$

Abbiamo così dimostrato in modo inconfutabile che applicare il trucchetto informatico equivale esattamente a calcolare l'inverso additivo $-x$ nella struttura ad anello dell'hardware. La magia non esiste, è solo ottima algebra!

Appendice Visiva (Esercizio Extra 2): La "Scalinata" della Precisione

Cosa stiamo guardando?
Il grafico illustra la "risoluzione" dell'universo IEEE-754. Sull'asse delle ascisse abbiamo un numero reale x, mentre sull'asse delle ordinate misuriamo l'ULP (*Unit in the Last Place*), ovvero l'esatta distanza matematica per raggiungere il **primissimo numero rappresentabile successivo**.



Perché una scalinata e non una curva morbida?
Questa visualizzazione distrugge l'intuizione del "continuo" reale. Nello standard a singola precisione (32 bit), la mantissa ha sempre e solo 23 bit di spazio. Pensate a questi 2^{23} (circa 8,3 milioni) di stati possibili come a delle "tacche" su un righello.

- **Nell'intervallo** $[1, 2)$: La lunghezza dell'intervallo è 1. Il calcolatore spalma i suoi 2^{23} stati uniformemente in questo spazio. La distanza tra una tacca e l'altra è minima e costante:
$$\Delta_1 = \frac{2-1}{2^{23}} = 2^{-23} \approx 0.000000119$$
Finché restiamo tra 1 e 2 (escluso), l'esponente hardware è bloccato a 2^0. Per questo la linea nel grafico è perfettamente orizzontale.

- **Il superamento del confine** ($x = 2.0$): Appena arriviamo a 2.0, l'esponente hardware deve necessariamente scattare a 2^1 per coprire il nuovo intervallo $[2, 4)$. Ma la lunghezza di questo nuovo intervallo è 2, il doppio del precedente! Avendo a disposizione **sempre e solo** 2^{23} tacche, per coprire una distanza doppia il calcolatore è costretto a distanziarle il doppio:
$$\Delta_2 = \frac{4-2}{2^{23}} = \frac{2}{2^{23}} = 2^{-22} \approx 0.000000238$$

L'illusione del continuo e il salto del bit meno significativo:
Potremmo essere tentati di pensare che per spostarci al numero successivo basti sempre aggiungere la stessa quantità infinitesima, variando l'ultimo bit della mantissa (LSB). La realtà dell'hardware smentisce questa intuizione tipica dell'analisi standard.

Se siamo fermi sul numero 1.0 (codificato come $1.000\ldots0_2 \times 2^0$), "accendere" il bit meno significativo aggiunge un valore pari a 2^{-23}. Il numero rappresentabile immediatamente successivo è quindi $1.0 + 2^{-23}$.

Tuttavia, non appena raggiungiamo il valore esatto 2.0, la normalizzazione impone il cambio di esponente: il numero viene codificato come $1.000\ldots0_2 \times 2^1$. Se da questa nuova posizione flippiamo *lo stesso identico* bit meno significativo della mantissa, il **peso assoluto** di quello

scatto viene inesorabilmente moltiplicato per la nuova potenza:

$$(1 + 2^{-23}) \times 2^1 = 2.0 + 2^{-22}$$

Il primissimo numero fisicamente esistente nel calcolatore dopo il 2 è esattamente $2 + 2^{-22}$. Qualsiasi numero reale compreso strettamente nell'intervallo aperto $(2.0,\ 2.0 + 2^{-22})$ cade nel vuoto tra queste due tacche hardware e non possiede alcuna rappresentazione. Questa perdita di risoluzione può sembrare trascurabile per numeri vicini all'unità (dove il salto è impercettibile all'occhio umano), ma genera grossi salti quando lavoriamo con ordini di grandezza elevati. Se, ad esempio, ci troviamo a esplorare la retta reale intorno a 2^{120}, il cambio di stato dell'LSB corrisponde a un incremento reale di:

$$(1 + 2^{-23}) \times 2^{120} = 2^{120} + 2^{97}$$

Il numero rappresentabile immediatamente successivo a 2^{120} dista la bellezza di 2^{97} unità (un valore astronomico, nell'ordine di grandezza di 10^{29}). Su queste scale immensamente grandi, la retta dei numeri smette del tutto di assomigliare a una linea continua e diventa un arcipelago di punti sparsi, separati da veri e propri "oceani" vuoti.

Appendice Visiva (Esercizio Extra 4): Il Frattale Twindragon e le Basi Complesse

Oltre la retta reale:
Nell'esercizio precedente abbiamo visto come l'utilizzo di una base negativa (-2) ci permetta di mappare l'intero insieme $\mathbb{Z}$ (numeri positivi e negativi) senza l'uso di bit di segno o isomorfismi come il complemento a 2. Ma cosa accade se estendiamo il concetto di "base posizionale" al campo dei numeri complessi $\mathbb{C}$?

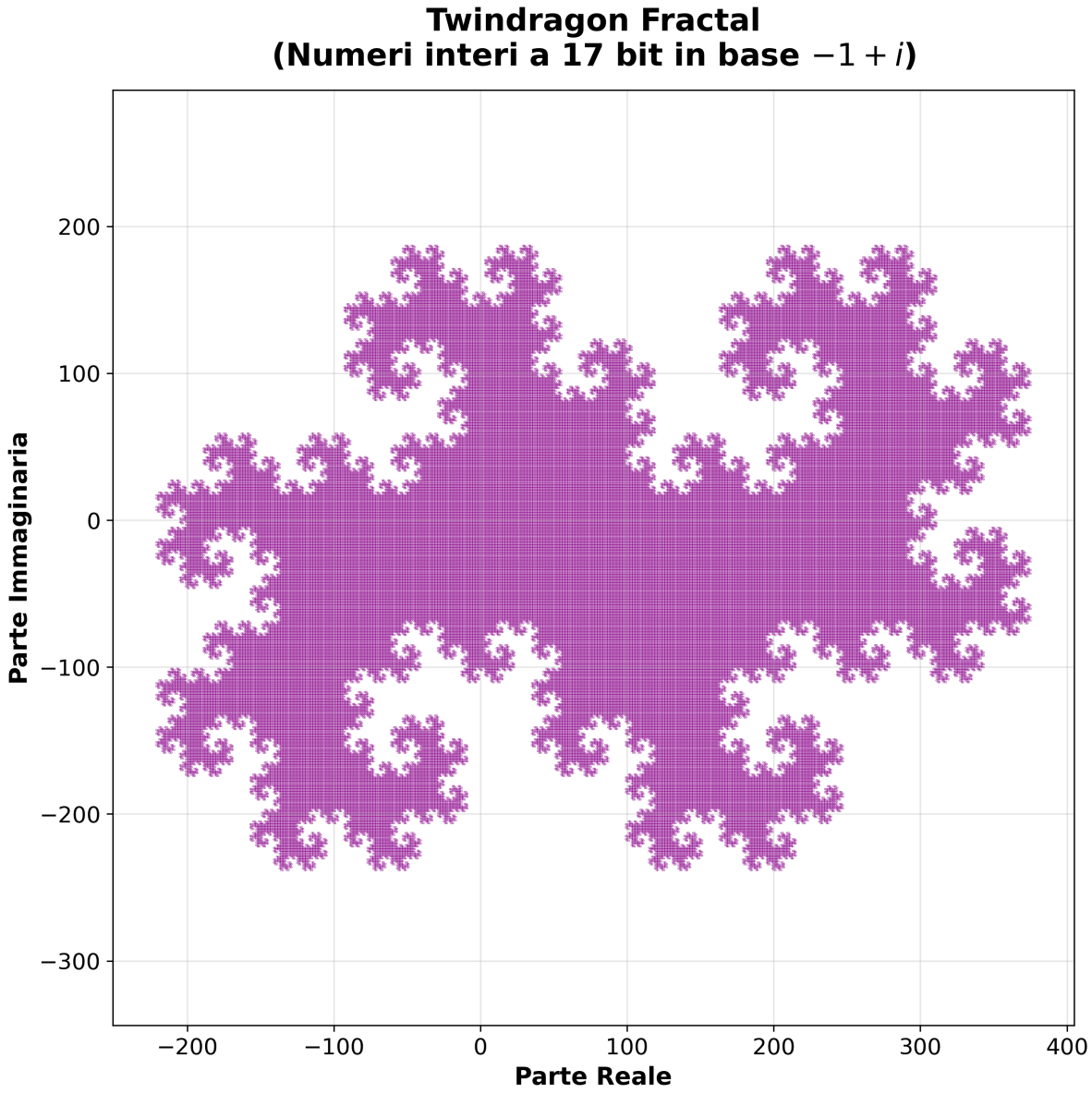
Il Sistema di Numerazione Complesso:
Consideriamo un sistema posizionale in cui la base sia il numero complesso $b = -1 + i$, mantenendo come alfabeto delle cifre solamente l'insieme canonico $\{0, 1\}$. Un numero in questo sistema è espresso dal polinomio:

$$Z = \sum_{k=0}^{n-1} c_k(-1 + i)^k \quad \text{con } c_k \in \{0, 1\}$$

La Geometria dell'Algebra:
Se in base 2 o in base -2 le combinazioni dei bit "accendono" punti disposti lungo una retta unidimensionale, l'aritmetica con base $-1+i$ produce una moltiplicazione complessa che implica rotazioni e dilatazioni sul piano di Argand-Gauss.

Se plottiamo sul piano complesso **tutti** i numeri rappresentabili con una stringa di 16 bit, i punti non si dispongono casualmente, né formano una griglia regolare. Essi convergono a formare un dominio strettamente connesso con una frontiera frattale, noto in topologia come **Twindragon Curve** (Curva del Drago Gemello).

Questa visualizzazione dimostra un fatto matematico profondo: i sistemi di numerazione posizionali non sono solo "modi per scrivere numeri", ma sono intrinsecamente legati alla geometria e possono essere utilizzati come sistemi di coordinate capaci di piastrellare in modo univoco spazi multi-dimensionali.


Twindragon Fractal
(Numeri interi a 17 bit in base $-1+i$)
Parte Immaginaria
Parte Reale
200
100
0
−100
−200
−300
−200
−100
0
100
200
300
400