Algoritmi Numerici e Strumenti di Calcolo

SUPSI Dipartimento Tecnologie Innovative

Gianni Grasso

15 novembre 2024

Classe: I1B

Anno scolastico: 2024/2025

Indice

1 Introduzione

La complessità (o efficenza) di un algoritmo è la quantità di risorse necessaria per la risoluzione di un determinato problema. Le risorse possono essere quantificate in termini di memoria o **tempo computazionale**.

La complessità temporale viene solitamente misurata contando il numero di operazioni elementari che devono essere svolte dall'algoritmo e viene espressa tramite la **notazione O-grande**, che si concentra sul termine di grado maggiore, escludendo i coefficienti e i termini di grado inferiore. Ad esempio, supponiamo che un determinato algoritmo richieda di svolgere $2n^3 + 4n^2 + n$, la sua complessità temporale è definita come $O(n^3)$ perchè il termine maggiore è n^3.

Ecco altri esempi di complessità temporali, dalla minore alla maggiore:

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

Facciamo un esempio pratico, scriviamo un algoritmo per sommare i primi n numeri interi partendo da 1. Il metodo più intuitivo sarebbe di rappresentarlo come:

$$S_n = \sum_{i=1}^{n} i = 1 + 2 + ... + n$$

questo algoritmo richiede di svolgere $n-1$ addizioni, la sua complessità temporale è quindi $O(n)$, ossia lineare.

Proviamo ora a trovare un algoritmo che faccia la medesima cosa ma con una complessità minore, consideriamo

$$S_n = \frac{n(n+1)}{2}$$

la cui validità può essere dimostrata matematicamente per induzione, non verrà dimostrata in questo documento.

Proviamo ad applicare questa seconda formula ai casi $n = 100$ e $n = 1000$:

$$S_{100} = \frac{100 + 101}{2} = 5050$$

$$S_{1000} = \frac{1000 + 10001}{2} = 500500$$

Notiamo che, in entrambi i casi, abbiamo dovuto effettuare solo tre operazioni distinte (un'addizione, una moltiplicazione e una divisione). Possiamo concludere che la complessità temporale di questo algoritmo è $O(1)$, ossia è costante e **indipendente dal valore di n**, a prescindere da quanto esso sia grande, preferiamo quindi usare il secondo algoritmo rispetto al primo.

2 Rappresentazione dei numeri

2.1 Algoritmo di Horner

2.1.1 Valutazione di un polinomio

Per capire Horner dobbiamo prima fare degli step intermedi che dimostrano la sua effettiva utilità, consideriamo un generico polinomio di terzo grado:

$$P(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

dove $a_0, ..., a_3$ sono coefficienti reali qualsiasi.

Calcoliamo quante operazioni sono necessarie per calcolare il valore del polinomio $P(x)$ in un generico punto x_0 effettuando le moltiplicazioni e le addizioni richieste:

$$P(x_0) = \underbrace{a_3x_0^3}_{\text{3 molt.}} + \underbrace{a_2x_0^2}_{\text{2 molt.}} + \underbrace{a_1x_0}_{\text{1 molt.}} + a_0$$

notiamo che sono necessarie esattamente 6 moltiplicazioni e 3 addizioni.

Ora invece consideriamo un generico polinomio di grado n:

$$P(x) = \sum_{k=0}^{n} a_kx^k = a_0 + a_1x + ... + a_nx^n$$

dove $a_0, ..., a_n$ sono coefficienti reali qualsiasi.

Abbiamo quindi lo stesso scenario del caso precedente ma con un polinomio di grado n invece che di grado 3. Procediamo in modo analogo a quanto fatto prima per determinare il numero di operazioni necessarie per calcolare il valore del polinomio x_0:

$$P(x_0) = \underbrace{a_nx_0^n}_{\text{n molt.}} + \underbrace{a_{n-1}x_0^{n-1}}_{\text{n - 1 molt.}} + ... + \underbrace{a_1x_0}_{\text{1 molt.}} + a_0$$

Sfruttando la formula introdotta nello scorso capitolo risulta che servono:

$$\underbrace{\frac{n(n+1)}{2} + n}_{\text{molt.}} = \frac{1}{2}n^2 + \frac{3}{2}n$$

operazioni, che corrispondono a una complessità temporale **quadratica**, ossia $O(n^2)$, poiché il termine maggiore è n^2.

Se ora consideriamo nuovamente il polinomio di terzo grado di prima possiamo notare che esso può essere riscritto come:

$$P(x) = a_3x^3 + a_2x^2 + a_1x + a_0 = ((a_3x + a_2)x + a_1)x + a_0$$

la cui correttezza può essere verificata sviluppando i conti. Notiamo che il numero di operazioni è diminuito, prima erano 6 moltiplicazioni e 3 addizioni mentre con la nuova formula le moltiplicazioni sono 3.

Questa formula può essere generalizzata ad un polinomio di grado n:

$$P(x_0) = (((a_nx_0 + a_{n-1})x_0 + a_{n-2}) \cdot ...)x_0 + a_0$$

che corrisponde all'**algoritmo di Horner**, che consente di risolvere un polinomio mediante n moltiplicazioni e n addizioni (come visto prima). In totale sono quindi necessarie $2n$ operazioni e ciò corrisponde a una complessita temporale **lineare**, ossia $O(n)$.

2.1.2 Applicazione

L'algoritmo di Horner è utile per passare da un sistema di numerazione qualsiasi al sistema decimale (ad esempio da binario a decimale).

Se consideriamo ad esempio il numero 3152, possiamo scriverlo come:

$$\begin{aligned} 3152 &= 3 \cdot 1000 + 1 \cdot 100 + 5 \cdot 10 + 2 \\ &= 3 \cdot 10^3 + 1 \cdot 10^2 + 5 \cdot 10^1 + 2 \cdot 10^0 \end{aligned}$$

A ciascuna cifra viene attribuita una diversa potenza di 10 in base alla posizione che occupa all'interno del numero considerato. Un generico numero $x \in \mathbb{N}$ può essere scritto come:

$$x = a_n 10^n + a_{n-1} 10^{n-1} + ... + a_0 = \sum_{i=0}^{n} a_i 10^i$$

dove $a_0, ..., a_n$ corrisponde alle cifre del numero.

Possiamo notare che la scrittura riportata è analoga a quella vista nella dimostrazione, che possiamo quindi riscrivere il numero come:

$$x = (((a_n \cdot 10 + a_{n-1}) \cdot 10 + a_{n-2}) \cdot ...) \cdot 10 + a_0$$

Per quanto riguarda gli altri sistemi numerici vale lo stesso discorso, consideriamo quindi un numero in una qualsiasi base b e scriviamolo nel corrispondente in base 10:

$$x = a_n b^n + a_{n-1} b^{n-1} + ... + a_0 = \sum_{i=0}^{n} a_i b^i$$

Ad esempio se vogliamo convertire 1201_3 in base 10, svolgiamo questi calcoli:

$$\begin{aligned} 1201_3 &= \sum_{i=0}^{3} a_i 3^i \\ &= 1 \cdot 3^3 + 2 \cdot 3^2 + 0 \cdot 3^1 + 1 \cdot 3^0 \\ &= 27 + 18 + 1 = 46_{10} \end{aligned}$$

Che utilizzando l'algoritmo di Horner diventa:

$$\begin{aligned} 1201_3 &= ((1 \cdot 3 + 2) \cdot 3 + 0) \cdot 3 + 1 \\ &= (5 \cdot 3 + 0) \cdot 3 + 1 \\ &= 15 \cdot 3 + 1 = 46_{10} \end{aligned}$$

2.1.3 Horner inverso

L'algoritmo di Horner inverso, o **algoritmo del modulo**, consente invece di passare da base 10 a base b.

2.2 Complemento a due

Vogliamo ora ampliare i numeri rappresentabili, grazie all'**algoritmo di Horner** infatti possiamo rappresentare solo i numeri naturali $\mathbb{N} = \{0, 1, 2, 3, 4, ...\}$. Ci serve quindi un modo per rappresentare anche i numeri interi $\mathbb{Z} = \{..., -3, -2, -1, 0, 1, 2, 3, ...\}$, ovvero il complemento a due. Gli insiemi interi positivi e negativi sono indicati rispettivamente con $\mathbb{Z}^+, \mathbb{Z}^-$.

Il complemento a due è in grado di rappresentare i numeri interi tramite l'uso delle seguenti regole:

1. il primo bit indica sempre il segno del numero (**0 positivo** e **1 negativo**)
2. se il primo bit è 0, il numero viene letto in base 10 usando $Horner(bit)$
3. se il primo bit è 1, il numero viene letto in base 10 come $-(2^n - Horner(bit))$

Facciamo alcuni esempi per quanto riguarda l'ultimo caso:

$$\begin{aligned}&1100_2\text{, il primo bit è 1, segno negativo}\\&Horner(1100) = (1 \cdot 2 + 1) \cdot 2 + ... = 12\\&-(2^4 - Horner(1100)) = -(16 - 12) = -4_{10}\end{aligned}$$

un altro esempio:

$$\begin{aligned}&1011 = 11_{10}\\&-(2^4 - 11) = -(16 - 11) = -5\end{aligned}$$

Consideriamo ora anche l'**algoritmo di Horner inverso**, che ci permette di passare da un numero intero alla sua rappresentazione in binario. Abbiamo quindi due casi distinti:

1. se il numero $x \in \mathbb{Z}_0^+$, applichiamo l'algoritmo inverso
2. se il numero $x \in \mathbb{Z}_0^-$, dobbiamo risolvere la seguente equazione per i bit:

$$\begin{aligned}&x = -(2^n - Horner(bit))\\\iff &x = -2^n + Horner(bit)\\\iff &Horner(bit) = 2^n + x\\\iff &bit = invHorner(2^n + x)\end{aligned}$$

dove *invHorner* indica l'inverso dell'algoritmo di Horner e n il numero di bit da cui è composta la memoria.

2.2.1 Interpretare le consegne

Durante lo svolgimento degli esercizi verranno menzionati spesso `short int` e `unsigned short int` o simili. Possiamo descriverli in questo modo:

- **Complemento a due**
  - short int 16 bit
  - int 32 bit
  - long int 64 bit
- **Unsigned (come numero naturale)**
  - short int 16 bit
  - int 32 bit
  - long int 64 bit

2.3 Horner su numeri razionali

Vogliamo ampliare ulteriormente i numeri rappresentabili. Fino ad ora pssiamo rappresentare tutti i numeri interi $\mathbb{Z} = ..., -3, -2, -1, 0, +1, +2, +3, ...$ Ci serve quindi un modo per rappresentare i numeri razionali, per farlo possiamo utilzare **Horner** sulla parte decimale di un numero.

Per convertire numeri razionali da base 10 a base b possiamo usare sempre l'algoritmo di Horner, con alcuni accorgimenti:

1. separiamo la parte intera dalla parte decimale
2. per la parte intera, usiamo l'algoritmo di Horner normalmente
3. per la parte decimale, usiamo una versione modificata che utilizza la funzione `INT()` per calcolare la parte intera di un numero

Ecco un esempio da base 10 a base 2:

$$\begin{aligned} int(0.1875 * 2) =& 0(MSB) \\ int(0.375 * 2) =& 0 \\ int(0.75 * 2) =& 1 \\ int(0.5 * 2) =& 1(LSB) \\ \Rightarrow 0.1875_{10} =& 0.0011_2 \end{aligned}$$

e uno da base 2 a base 10, in questo caso si parte dal LSB:

$$\begin{aligned} 1 \longrightarrow \frac{0}{2} + 1 =& 1 \\ 1 \longrightarrow \frac{1}{2} + 1 =& 1.5 \\ 0 \longrightarrow \frac{1.5}{2} + 0 =& 0.75 \\ 0 \longrightarrow \frac{0.75}{2} + 0 =& 0.375 \\ \longrightarrow \frac{0.375}{2} =& \ 0.1875 \end{aligned}$$

2.4 Horner su numeri reali

Al punto in cui siamo arrivati ora non abbiamo modo di esprimere delle radici con Horner. Vogliamo dunque ampliare l'insieme razionale $\mathbb{Q}$ ed estenderlo a tutti i numeri reali $\mathbb{R}$.

Per evitare errori di approssimazioni utilizziamo la notazione scientifica, consideriamo però una notazione scientifica in base 2 e non in base 10 come siamo abituati.

Per farlo possiamo usare il formato float, dati 32 bit esso si basa sulla seguente allocazione:

1. il primo bit per il segno (1 se negativo, 0 se positivo)
2. 8 bit per l'esponente
3. i rimanenti 23 bit per la mantissa (1.XXX)
4. infine la base (della notazione scientifica) è sempre pari a 2

Nota: è sempre possibile scrivere il numero da rappresentare nella forma della mantissa.

Nota: il formato float non utilizza il complemento a due.

Con la notazione appena fornita non possiamo rappresentare lo zero, i numeri più vicini allo zero che possiamo rappresentare sono

$$
\begin{aligned}
+1.0 * 2^{-127} &= -\frac{1}{127} \\
-1.0 * 2^{-127} &= \frac{1}{127}
\end{aligned}
$$

entrambi diversi da zero.

2.5 Applicazione

- Esponente, $exp = horner(bit) - 127$
- Numero, $n = sgn * 1.XXX * 2^{exp}$

2.5.1 Numeri macchina

C'è però un problema, i numeri reali sono infiniti, mentre i numeri rappresentabili da un computer sono finiti. Dobbiamo quindi scegliere dei numeri macchina, ovvero i valori che possiamo effettivamente rappresentare con 23 bit, i valori che non sono numeri macchina vengono aapprossimati al numero macchina più vicino.

Se ad esempio un numero intero non è rappresentabile, troveremo il numero macchina più vicino ad esso e, sottraendolo al numero di partenza troveremo l'errore assoluto di quella rappresentazione.

- Errore assoluto
  - x: numero reale
  - x_m: numero macchina

  $$
  \begin{aligned}
  e_a &= |x - x_m| \\
  e_a &< 2^{p-s-1}
  \end{aligned}
  $$

  dove nell'ultima formula p è l'esponente del numero e s il numero di bit della mantissa.

- Errore relativo

  $$
  \begin{aligned}
  e_r &= \frac{e_a}{x} \\
  e_r &= \frac{|x - x_m|}{x}
  \end{aligned}
  $$

3 Risoluzione numerica di equazioni non lineari

In questa sezione verranno trattati i metodi di bracketing, dei metodi che, dati due intervalli di una funzione, trovano una soluzione approssimata di tale funzione.

3.1 Metodo di bisezione

Il metodo di bisezione è un metodo di ricerca incrementale in cui un intervallo che contiene uno zero della funzione viene ripetutamente dimezzato per localizzare con maggior precisone il valore esatto.

Considerando quindi una funzione $f : \mathbb{R} \longrightarrow \mathbb{R}$ che sia continua possiamo eseguire i seguenti passaggi:

- Scegliere due numeri a e b tali che

$$f(a) \cdot f(b) < 0$$

- Approssimare lo zero della funzione con il punto medio dell'intervallo $[a, b]$

$$c = \frac{a+b}{2}$$

- Determinare in quale intervallo si trova il risultato valutando $f(a) \cdot f(c)$
  - Se $f(a) \cdot f(c) < 0$: impostare $b = c$ e ripetere
  - Se $f(a) \cdot f(c) > 0$: impostare $a = c$ e ripetere
  - Se $f(a) \cdot f(c) = 0$: terminare il processo (lo zero è esattamente c)

3.1.1 Tolleranza

Il secondo e il terzo passaggio sono da ripetere finchè non si trova lo zero esatto oppure, scenario più probabile, fino a che $f(c)$ sarà un valore sufficentemente vicino a zero secondo i nostri criteri.

Possiamo infatti definire una certa soglia di tolleranza sotto la quale consideriamo il risultato esatto, in questo caso procediamo come segue:

- Definiamo un valore $\epsilon = 0.00...01$
- Controlliamo ad ogni iterazione che

$$|f(c)| < \epsilon$$

  se questa condizione è soddisfatta ci fermiamo, altrimenti ripetiamo il procedimento

Possiamo inoltre definire il numero di iterazioni che l'algoritmo impiegherà per raggiungere tale precisione:

$$\begin{aligned} &\frac{|b-a|}{2^{n+1}} \leq \epsilon \\ &\iff n \geq \frac{log(b-a) - log(\epsilon)}{log(2)} - 1 \end{aligned}$$

3.2 Regula falsi

La regula falsi (o False Position) è un metodo alternativo al metodo di bisezione per trovare lo zero di una funzione. Utilizzando questo metodo non viene fatta la media dei punti a e b per trovare c, viene traccaita una retta tra i due punti e c diventa il punto che interseca l'asse delle x su quella retta. Questo ci fa risparmiare tempo e operazioni in determinati casi.

- Scegliere due numeri a e b tali che

$$f(a) \cdot f(b) < 0$$

- Calcoliamo c come punto della retta ab che interseca l'asse x

$$c = a - \frac{f(a)}{\frac{f(b)-f(a)}{(b-a)}}$$

- Determinare in quale intervallo si trova il risultato valutando $f(a) \cdot f(c)$
  - Se $f(a) \cdot f(c) < 0$: impostare $b = c$ e ripetere
  - Se $f(a) \cdot f(c) > 0$: impostare $a = c$ e ripetere
  - Se $f(a) \cdot f(c) = 0$: terminare il processo (lo zero è esattamente c)

Ci sono alcuni casi però in cui questa metodo risulta sconveniente rispetto al metodo di bisezione. Per risolvere questo problema possiamo combinare i due metodi, si parte quindi con un iterazione di regula falsi seguito da una o più interazioni del metodo di bisezione. Per determinare quando passare di nuovo al metodo di bisezione guardiamo dove l'intervallo rimasto uguale durante la prima iterazione cambia.