COMPLESSITÀ COMPUTAZIONALE

COMPLESSITÀ COMPUTAZIONALE DI UN ALGORITMO

- = Misura **assoluta** delle **risorse** necessarie alla sua esecuzione
- Assoluta
  - non dipende da chi lo esegue (potenza computer)
- Risorse
  - analisi della **complessità temporale**
  - analisi della **complessità spaziale**

COME MISURARE PRESTAZIONI?

- Usare cronometro non è la soluzione migliori
  - parte del tempo reale non dipende dall'algoritmo
- Utilizzo metodo `System.currentTimeMillis()`
  - numero di millisecondi da evento riferimento (01/01/1970)
  - facendo la differenza tra le due chiamate del metodo, si trova tempo di esecuzione real
- Eseguire algoritmo con array di dimensioni (n) diverse
  - ripetere misura + volte per trovare valore medio $T(n)$
- Plottare risultato su piano cartesiano
  - individuare curva che approssima i dati

ANALISI TEORICA DELLE PRESTAZIONI

- Modello di costo di un algoritmo che dipende:
  - numero di **operazioni primitive** (passi base)
  - **dimensione dei dati** da elaborare
  - **valore dei dati**
- Analisi senza **realizzare** e **compilare** un algoritmo
  - senza programmarlo
- Non deve dipendere dalla potenza del computer

OPERAZIONI PRIMITIVE

- Definita anche passo base
- Operazione che ha **tempo di esecuzione costante**
  - NON dipende dall'input

- Esempi
  - assegnazione valore a una variabile
  - operazione aritmetica/logica tra variabili primitive
  - accesso in lettura/scrittura a un elemento di un array
  - Valutazione espressione booleana
  - NO: invocazione di un metodo ricorsivo

DIMENSIONE DEI DATI

- A seconda dell'input, assume significati diversi:
  - **grandezza di un numero**
    - in calcolo numerico
  - **numero di elementi**
    - problemi di ordinamento con array
  - **numero di bit** di un numero
- n = dimensione input

VALORE DEI DATI

- Durata esecuzione dipende da valore dei dati
  - se contiene cicli e decisioni
- Metodi di stima:
  - stima di **caso peggiore**
  - stima di **caso migliore**
  - stima di **caso medio**
    - es. numeri casuali

BIG-O NOTATION

- Si ottiene considerando soltanto il termine che si incrementa più rapidamente al variare di n, ignorando coefficienti costanti
- $f(n) = O(g(n))$
  - $f(n)$ cresce in egual modo o più lentamente di $g(n)$
  - relazione con gli o-piccolo
    - $f(n) = o(g(n)) \vee f(n) \sim l * g(n) \text{ per } n \to \infty$
  - es: $f(n) = \frac{1}{2}n^2 = O(n^2) = O(n^3)$
- $f(n) = \Omega(g(n))$
  - $f(n)$ ==cresce in egual modo o più velocemente di $g(n)$
- $f(n) = \Theta(g(n))$
  - $f(n)$ ==cresce con la stessa velocità di $g(n)$

ORDINI DI COMPLESSITÀ

<u>CLASSIFICAZIONE ALGORITMO</u>

- In funzione delle prestazioni
  - Efficiente: al massimo **polinomiale**
  - Inefficiente: almeno **esponenziale**

<u>CLASSIFICAZIONE PROBLEMA ALGORITMICO</u>

- In funzione del **più veloce** algoritmo che lo risolve
  - Trattabile: complessità al massimo **polinomiale**
  - Non trattabile: complessità almeno **esponenziale**

- Ipotesi: una istruzione di Java viene eseguita in *10^{-7} s*

(1 anno ≈ 3E7 secondi

(convenzione anglosassone: 1 “bilione” = 10^9; 1 “trilione” = 10^{12}

	f(n)	n=10	n=20	n=50	n=100	n=300
Problemi trattabili (polinomiali)	n^2	1/100000 secondi	1/25000 secondi	1/4000 secondi	1/1000 secondi	9/1000 secondi
	n^5	1/100 secondi	0.32 secondi	31.2 secondi	16.7 minuti	2.8 giorni
Problemi non trattabili	2^n	1/10000 secondi	0.1 secondi	3.57 anni	40 trilioni di secoli	Un numero a 74 cifre di secoli
	n^n	16.7 minuti	332 bilioni (miliardi) di anni	Un numero a 69 cifre di secoli	Un numero a 184 cifre di secoli	Un numero a 727 cifre di secoli

ANALISI PRESTAZIONI - SELECTION SORT

<u>ANALISI NUMERO ACCESSI LETTURA/SCRITTURA AL VARIARE DI N</u>

- **Conteggio accessi** all'array nella prima iterazione del ciclo esterno (i=0)
  - per trovare elemento minore: n accessi
  - 4 accessi per swap
    - caso peggiore: serve sempre effettuare lo swap
  - $(n+4)$ accessi in totale
- Ora deve ordinare parte rimanente di $(n-1)$ elementi
  - $[(n-1)+4]$ accessi

- Si arriva fino al passo con 2 elementi
- Totale:
  - $T(n) = (n+4) + ((n-1)+4) + ((n-2)+4) + \ldots + (2+4)$
    $= [n + (n-1) + (n-2) + \ldots + 3 + 2] + (n-1) * 4$
    $= \frac{n(n+1)}{2} - 1 + (n-1) * 4$
    $= \frac{1}{2}n^2 + \frac{9}{2}n - 5$
  - si ottiene andamento parabolico (come trovato sperimentalmente)

ANDAMENTO ASINTOTICO PER VALORI ELEVATI DI N

- È utile individuare le prestazioni per **valori elevati di n**
  - si studia andamento asintotico
  - $T(n) \sim \frac{1}{2}n^2$
- Applicando un'ulteriore semplificazione e ingnorando i fattori costanti:
  - $T(n) = c\,n^2$
  - da questa relazione si capisce che se n raddoppia, il tempo di esecuzione quadruplica
- $O(n^2)$ = numero di accessi è dell'ordine di n^2

STIMA COMPLESSITÀ ALGORITMO CON DUE CICLI ANNIDATI

```
for (int i = 0; i < n; i++) {
    //... operazioni primitive
    for (int j = i; j < n; j++) {
        //... operazioni primitive
    }
}
```

- Numero totale: $\sum_i^n i = \frac{n(n+1)}{2} = O(n^2)$
  - cicli annidati di questo tipo hanno sempre prestazioni *O(n^2)*

> **Domanda**
>
> Se si aumenta di 10 volte la dimensione dei dati, come aumenta il tempo richiesto per ordinare dati usando selectionSort?

ANALISI PRESTAZIONI - MERGE SORT

ANALISI DELLE SINGOLE FASI

- Creazione dei due sottoarray
  - $2n$ accessi
    - tutti gli elementi devono essere letti e scritti

- Invocazioni ricorsive
  - $T(n/2)$
    - contengono metà elementi
- Fusione
  - $2n$
    - per ogni elemento che si andrà a scrivere nell'array finale, bisogna leggere due elementi (per confrontarli), uno da ciascun array da fondere
  - n
    - accessi nella scrittura dell'array finale

TOTALE

- $T(n) = 2T(\frac{n}{2}) + 5\,n$
  - equazione per **ricorrenza**
- Si procede per sostituzioni successive
  - $T(\frac{n}{2}) = 2T(\frac{n}{4}) + 5\,\frac{n}{2}$
  - $T(\frac{n}{4}) = 2T(\frac{n}{8}) + 5\,\frac{n}{4}$
  - ...
  - $T(1) = 1$
- Facendo la somma totale
  - $T(n) = 2T(\frac{n}{2}) + 5n$
    $= 2(2T(\frac{n}{4}) + \frac{5}{2}n) + 5n$
    $= 2^2\,T(\frac{n}{2^2}) + 2 \cdot 5\,n$
    $= \ldots$
    $= 2^k\,T(\frac{n}{2^k}) + k * 5\,n$
- Si raggiunge caso base quando:
  - $\frac{n}{2^k} = 1 \Leftrightarrow k = \log_2 n$
- Sostituendo il k trovato:
  - $2^k\,T(\frac{n}{2^k}) + k \cdot 5\,n = n * 1 + 5n * \log_2 n = O(n \log n)$
- In conclusione: $T(n) = O(n \log n)$
  - più veloce di selection sort

STIMA COMPLESSITÀ ALGORITMO DEFINITO PER RICORRENZA

- $T(n) = aT(n/b) + f(n)$
- Si può calcolare
  - per sostituzione
  - utilizzando Master Theorem

ANALISI PRESTAZIONI - INSERTION SORT

INSERTION SORT SU ARRAY NON ORDINATI

- Array n elementi
- Ciclo esterno: $n-1$ iterazioni (parto da 1)
- A ogni iterazione
  - 2 accessi (1 in lettura prima del ciclo e 1 prima in scrittura)
  - ciclo interno
    - 3 accessi per ogni elemento a sinistra
- Caso peggiore: (dati ordinati al rovescio)
  - 2 accessi (lettura/scrittura) per ogni iterazione: $(n-1)$ volte
  - 3 accessi per ogni elemento da spostare a sinistra
    - $1+2+3+...+(n-1) = 3\sum_{k=0}^{n-1} k = 3\frac{(n-1)n}{2}$
  - totale
    - $T(n) = 2(n-1) + 3\frac{(n-1)n}{2} = O(n^2)$
- Caso migliore: (dati già ordinati)
  - per ogni iterazione: $(n-1)$ volte
    - 2 accessi (lettura/scrittura)
    - 1 accesso nel ciclo interno per verificare che la condizione sia verificata
  - totale:
    - $T(n) = 3 * (n-1) = O(n)$
- Caso medio: richiede in media lo spostamento di metà degli elementi alla sua sinistra
  - stimo metà accessi nel ciclo interno
  - totale
    - $T(n) = 2(n-1) + 3\frac{(n-1)n}{4} = O(n^2)$

CONFRONTO TRA ORDINAMENTI

- Se l'array è quasi ordinato, conviene "insertion sort" altrimenti "merge sort"
- Esempio notevole
  - un array che viene mantenuto ordinato per effettuare ricerche, inserendo ogni tanto un nuovo elemento e poi riordinandolo periodicamente

	caso migliore	caso medio	caso peggiore
merge sort	n lg n	n lg n	n lg n
selection sort	n^2	n^2	n^2
insertion sort	n	n^2	n^2

Prestazioni in sintesi

- **Selection sort**: $O(n^2)$
- **Merge sort**: $O(n \log n)$
- **Insertion sort**: $O(n^2)$ (caso peggiore) - $O(n)$ (caso migliore)
- **Insertion sort** su array quasi **ordinati** (eccetto ultimo elemento inserito): $O(n)$ (caso peggiore) - $O(1)$ (caso migliore)

ANALISI PRESTAZIONI - LINEAR SEARCH

- Devo sempre fare n accessi
- $T(n) = O(n)$

ANALISI PRESTAZIONI - BINARY SEARCH

- Algoritmo è ricorsivo
- $T(n) = T\left(\frac{n}{2}\right) + 1$
- Risolvo per **sostituzioni successive**
- $T(n) = T(\frac{n}{2}) + 1 = T(\frac{n}{4}) + 1 + 1 = \ldots = T\left(\frac{n}{2^k}\right) + k$
- Arrivo al caso base $T(1)$ quando $\frac{n}{2^k} = 1$ ovvero $k = \log_2 n \Leftrightarrow n = 2^k$
- $T(n) = T(1) + \log_2 n = 1 + \log_2 n = O(\log n)$

Domanda

Si immagini di cercare con **binarySearch** un numero telefonico in un array ***ordinato*** di 10000(
di dati. Quanti dati vanno esaminati mediamente per trovare il numero?

ANALISI PRESTAZIONI - ALGORITMO DI FIBONACCI RICORSIVO

- Ricorsione multipla ($F_n = F_{n-1} + F_{n-2}$)
- $T(n) = T(n-1) + T(n-2) > T(n-2) + T(n-2)$
  $= 2T(n-2) > 4T(n-4) > 2^k * T(n-2k)$
  - $T(1)$ si ha quando $n - 2k = 1$ ovvero $k = \frac{n-1}{2}$
  - $T(n) < 2^{\frac{n-1}{2}}$
- $T(n) = T(n-1) + T(n-2) < T(n-1) + T(n-1)$
  $= 2T(n-1) > 4T(n-2) > 2^k * T(n-k)$
  - $T(1)$ si ha quando $n - k = 1$ ovvero $k = n - 1$
  - $T(n) > 2^{n-1}$
- $2^{n-1} < T(n) < 2^{\frac{n-1}{2}} \implies T(n) = O(2^n)$

COMPLESSITÀ SPAZIALE

- Si considera il picco di occupazione di memoria
- Si distinguono
  - azioni che occupano **quantità costante**
    - variabili dati fondamentali
  - azioni che occupano **quantità variabile**
    - array e stringhe
  - **memoria ausiliaria**
    - occupata dal codice stesso
- È importante considerare anche **garbage collection**
  - quando si esce da un {blocco}, lo spazio delle variabili locali viene liberate
- Cicli iterativi
  - non accumulano spazio perché viene liberato dopo ogni iterazione
  - complessità spaziale $O(1)$
- Cicli ricorsivi
  - si possono impilare fino a n chiamate del metodo nel punto di picco
  - complessità spaziale $O(n)$

ANALISI DI UN PROBLEMA

- Invece di analizzare il codice, si può analizzare la complessità degli algoritmi che si utilizzano per risolverlo
- È importante valutare la complessità computazionale se in presenza di **input grandi**