



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica



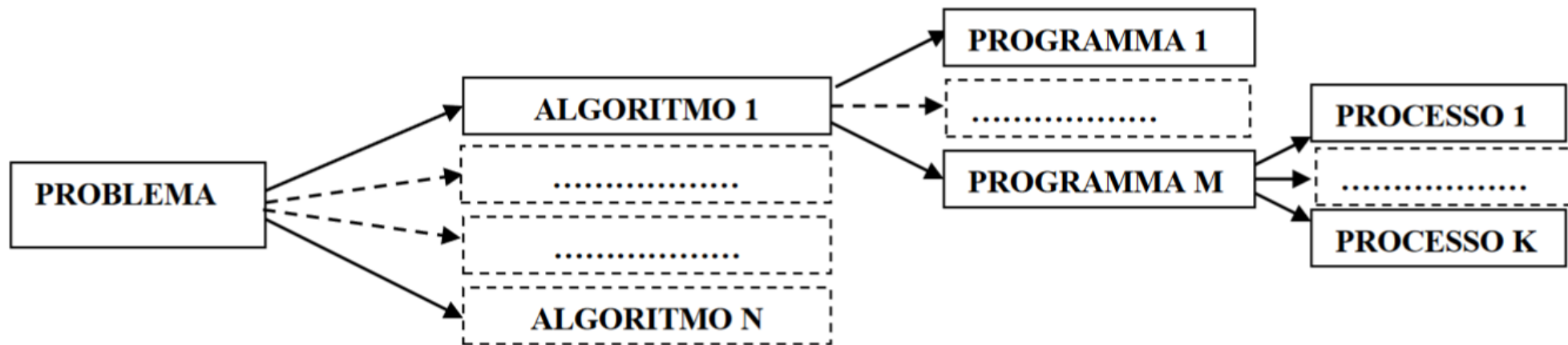
Laboratorio di Algoritmi e Strutture Dati a.a. 2024/2025

ALGORITMI E LORO IMPLEMENTAZIONE IN JAVA:
Introduzione

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM

Premessa

- Dato un problema, possono esistere **più algoritmi risolutivi** che sono **corretti** rispetto ad esso
 - e un numero illimitato di algoritmi errati :(
- Ogni **algoritmo** può essere tradotto in un **programma** scritto in un linguaggio di programmazione e tale programma verrà trasformato in un **processo** a tempo di esecuzione



Algoritmo

- Da un punto di vista computazionale, un algoritmo è una procedura che prende dei **dati in input** e, dopo averli elaborati, restituisce dei **dati in output**
- ⇒ I dati devono essere organizzati e strutturati in modo tale che la procedura che li elabora sia “parsimoniosa” (**efficiente**)
- ⇒ Il concetto di algoritmo è inscindibile da quello di dato

Complessità computazionale

- L'esecuzione di un algoritmo su un dato input richiede **risorse di tempo e di spazio** il cui ammontare prende il nome di **complessità computazionale**
- Un consumo eccessivo di risorse può pregiudicare la possibilità di utilizzo di un algoritmo
- È di fondamentale importanza saper trovare una **soluzione algoritmica efficiente e possibilmente ottimale**, a specifici problemi ben formalizzati.

Obiettivo

“Dato un problema, **trovare un algoritmo corretto e funzionante** che ne descriva il relativo procedimento risolutivo e codificarlo in un determinato linguaggio di programmazione”



“Dato un problema, **trovare tra i vari algoritmi risolutivi quello migliore possibile** confrontandoli dal punto di vista dell’«efficienza»

- Ogni algoritmo è caratterizzato da una **complessità temporale e spaziale** rispetto alle «dimensioni dei dati di ingresso» (concetto introdotto e approfondito in modo formale nel modulo di ASD)

Ciclo di sviluppo di codice algoritmico

- Lo sviluppo di software **robusto** ed **efficiente** per la soluzione di problemi di calcolo segue uno schema semplificato a due fasi che si avvicendano in un processo ciclico:
 - **Fase progettuale**
 - **Fase realizzativa**
- Richiede (tra le altre cose): capacità di astrazione, familiarità di strumenti matematici, padronanza del linguaggio di programmazione, creatività.

Fase progettuale (1 di 5)

- A. Si definiscono i **requisiti** del problema di calcolo che si intende affrontare:
- Definire in modo preciso e non ambiguo il problema di calcolo che si intende risolvere
 - Identificare i requisiti dei dati in ingresso e di quelli in uscita prodotti dall'algoritmo
 - Già in questa fase è possibile valutare se un problema complesso può essere **decomposto in sottoproblemi** risolvibili in modo separato e indipendente

Fase progettuale: problema di ordinamento

Definizione dei requisiti di un **problema di ordinamento**:

- **Input:** un insieme di elementi qualsiasi $A = \{a_1, \dots, a_n\}$ su cui sia possibile definire una relazione di ordine totale \leq (ossia una relazione riflessiva, antisimmetrica e transitiva definita su ogni coppia di elementi dell'insieme)
- **Output:** una permutazione degli elementi dell'insieme, in modo tale che $a_{i_h} \leq a_{i_k}$ per ogni $h \leq k$ ($h, k = 1, 2, \dots, n$)

Fase progettuale: problema di ricerca

Definizione dei requisiti di un **problema di ricerca**:

- **Input:** un insieme di elementi qualsiasi $A = \{a_1, \dots, a_n\}$ e un elemento k (chiave)
- **Output:** indice i tale che:
 - se $k \in A$, $i \in \{1, \dots, n\}$ e $a_i = k$
 - se $k \notin A$, $i = -1$

Fase progettuale (2 di 5)

B. Si studia la **difficoltà intrinseca del problema**, ossia la quantità minima di risorse di calcolo (tempo e memoria di lavoro) di cui **qualsiasi algoritmo** ha bisogno per risolvere una generica istanza del problema dato.

⇒ Per molti problemi importanti non sono ancora noti **limiti inferiori** precisi che ne caratterizzano la difficoltà intrinseca, per cui non è ancora possibile stabilire se un algoritmo risolutivo sia ottimo o meno

Fase progettuale (3 di 5)

- C. Si progetta un algoritmo risolutivo, verificandone formalmente la **correttezza** e stimandone le **prestazioni teoriche**
- La complessità dell'algoritmo viene espressa in funzione della dimensione n dell'istanza (**analisi asintotica**)
 - Tra i vari algoritmi risolutivi, l'obiettivo è trovare quello che faccia il miglior uso possibile delle risorse di calcolo disponibili (tempo di esecuzione ed occupazione di memoria)
 - Si valuta il tempo di esecuzione (in «**numero di passi**») in modo indipendente dalla tecnologia dell'esecutore

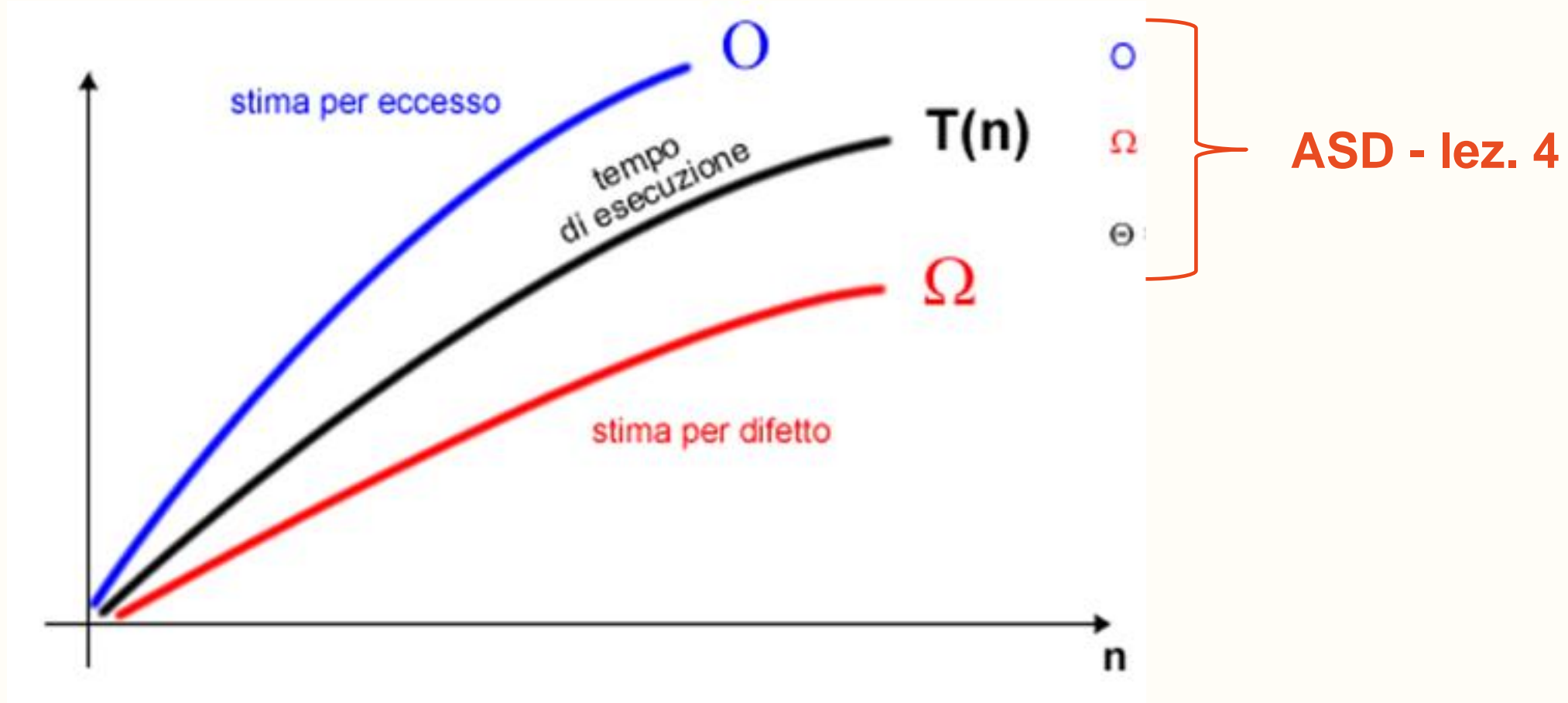
Fase progettuale (4 di 5)

- In grossi progetti software è fondamentale stimare le prestazioni già a livello progettuale.
- Scoprire solo dopo la codifica che i requisiti prestazionali non sono stati raggiunti potrebbe portare a conseguenze disastrose o per lo meno molto costose.

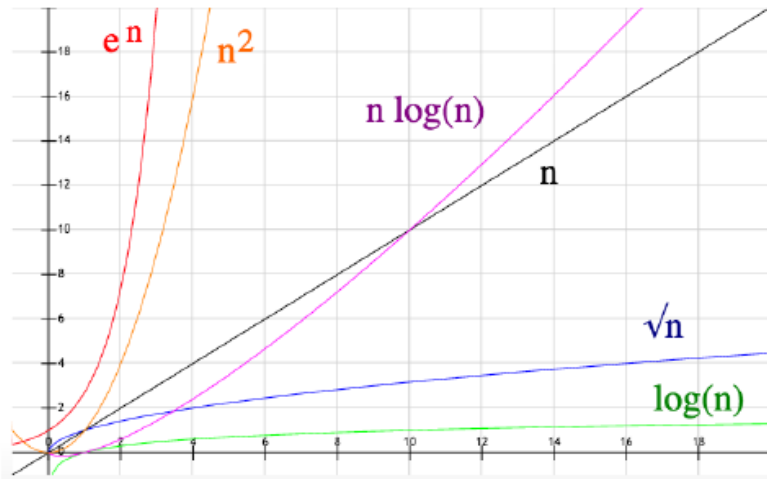
Fase progettuale (5 di 5)

- Il tempo/spazio di calcolo **necessario** alla risoluzione di un dato problema (**difficoltà intrinseca del problema**): la quantità minima di risorse di calcolo necessarie al caso peggiore **per ogni algoritmo** che risolve una generica istanza del problema dato
- Il tempo/spazio di calcolo **sufficiente** alla risoluzione di un dato problema: la quantità di risorse di calcolo necessarie al caso peggiore ad **uno specifico algoritmo** che risolve una generica istanza del problema dato
- Qualora la verifica della correttezza rilevi problemi o la stima delle prestazioni risulti poco soddisfacente si torna al passo C (se non al passo B...)

Delimitazione superiore e inferiore alla complessità di un problema (informale)



Tempi di esecuzione a confronto



n	$n/2$	$\log(n)$
10	5	3,321928
20	10	4,321928
30	15	4,906891
40	20	5,321928
50	25	5,643856
60	30	5,906891
70	35	6,129283
80	40	6,321928
90	45	6,491853
100	50	6,643856
300	150	8,228819
1000	500	9,965784
10000	5000	13,28771
100000	50000	16,60964

Fase realizzativa

- Si **codifica** l'algoritmo progettato in un linguaggio di programmazione e lo si **collauda** per identificare eventuali errori implementativi
- Si effettua un'**analisi sperimentale** del codice prodotto e se ne studiano le **prestazioni pratiche**
- Si ingegnerizza il codice, migliorandone la struttura e l'efficienza pratica attraverso opportuni accorgimenti
- Non è raro che l'analisi sperimentale fornisca suggerimenti utili per ottenere algoritmi più efficienti anche a livello teorico.

Il problema dei duplicati (A)

Formulato come un **problema di decisione**

Input: una sequenza S di elementi qualsiasi $S = \{s_1, \dots, s_n\}$

Output: **true** se esiste in S una coppia di elementi duplicati (cioè esiste in S una coppia di indici distinti $i, j \in \{1, \dots, n\}$ tale che $s_i = s_j$), **false** altrimenti.

Il problema dei duplicati (B)

- **Difficoltà intrinseca del problema $\Omega(n)$:** la delimitazione inferiore banale di ogni algoritmo è dell'ordine di grandezza di n (almeno la lettura dei dati in ingresso)

Il problema dei duplicati (C)

- Analisi della correttezza e del tempo di esecuzione di **verificaDup** per una generica istanza di dimensione n (per $n \rightarrow \infty$)

```
Algoritmo verificaDup (sequenza S)
    for each elemento x della sequenza S do
        for each elemento y che segue in S do
            if x=y then return true
    return false
```

verificaDup: correttezza

- L'algoritmo confronta almeno una volta ogni coppia di elementi, per cui se esiste un elemento che si ripete in S verrà sicuramente trovato.

verificaDup: complessità (1 di 3)

Stima delle prestazioni: “Quanto tempo richiede l’algoritmo?”

- La metrica deve essere indipendente dalle tecnologie e dalle piattaforme utilizzate (il numero di passi richiesto dall’algoritmo)
 - “Misuriamo il tempo in secondi?” La risposta cambierebbe negli anni o anche semplicemente su piattaforme diverse
- La metrica deve essere indipendente dalla particolare istanza (tempo espresso in funzione della dimensione n dell’istanza, **notazione asintotica**)
 - “Lo sforzo richiesto per analizzare 10 elementi e per analizzarne 1 milione è lo stesso?”

verificaDup: complessità (2 di 3)

- Informalmente, per valutare l'ordine di grandezza " $O(\cdot)$ " o tasso di crescita del **tempo di esecuzione** dell'algoritmo **verificaDup**, possiamo contare quanti confronti ("operazione dominante") si eseguono al crescere di n
 - **$O(1)$** (ordine di grandezza "costante") per istanze più favorevoli per l'algoritmo (caso migliore)
 - **$O(n*n)$** (ordine di grandezza "quadratico" di n^2) per istanze più sfavorevoli (**c.p.** - **caso peggiore**)

Calcolo della complessità in pratica (algoritmi non ricorsivi)

- Vengono “contate” le operazioni eseguite
- Il tempo di esecuzione di un’**istruzione di assegnamento** che non contenga chiamate a funzioni è 1
- Il tempo di esecuzione di una **chiamata ad una funzione** è $1 +$ il tempo di esecuzione della funzione
- il tempo di esecuzione di un’**istruzione di selezione** è il tempo di valutazione dell’espressione + il tempo massimo fra il tempo di esecuzione del ramo True e del ramo False
- il tempo di esecuzione di un’**istruzione di ciclo** è dato dal tempo di valutazione della condizione + il tempo di esecuzione del corpo del ciclo moltiplicato per il numero di volte in cui questo viene eseguito

Richiami: somme parziali

$$\sum_{k=0}^n r^k = \frac{1-r^{n+1}}{1-r} \quad \forall n \in \mathbb{N}, \forall r \in \mathbb{R}, r \neq 1$$

(progressione geometrica di ragione r)

$$\sum_{k=1}^n k = \frac{n(n+1)}{2} \quad \forall n \geq 1$$

(progressione aritmetica)

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6} \quad \forall n \geq 1$$

$$\sum_{k=1}^n k^3 = \left(\frac{n(n+1)}{6} \right)^2 \quad \forall n \geq 1$$

$$\sum_{k=1}^n \frac{1}{k(k+1)} = 1 - \frac{1}{n+1} \quad \forall n \geq 1$$

(somma telescopica)

$$\sum_{k=1}^n \sin k = \frac{\sin \frac{n}{2} \cdot \sin \frac{n+1}{2}}{\sin \frac{1}{2}}$$

verificaDup: correttezza e complessità

Algoritmo **verificaDup** (sequenza S)

```
for each elemento x della sequenza S do
    for each elemento y che segue in S do
        if x=y then return true
return false
```

- $T(n) = \sum_{k=1}^{n-1} k$ (progressione aritmetica)
- $T(n) = O(n^2)$
- Esistono algoritmi più efficienti di **verificaDup**?

verificaDup: complessità (3 di 3)

- Osserviamo che se la sequenza in ingresso è ordinata possiamo risolvere il problema più efficientemente
- gli eventuali duplicati sono in posizione consecutiva
- è sufficiente scorrere l'intera sequenza

Il problema dei duplicati (C-2)

Idea nuovo algoritmo:

- Ordinare la sequenza (*sarà argomento del corso!*)
 - $\theta(n \cdot \log n)$, ordine di grandezza pseudo-polinomiale
- Cercare due elementi duplicati consecutivi
 - $O(n)$ nel c.p., ordine di grandezza lineare
- tempo di esecuzione complessivo: $O(n \cdot \log n)$ nel c.p.

Il problema dei duplicati (C-2 continua)

Algoritmo verificaDupOrd (sequenza S)

ordina S in modo non-decrescente

for each elemento x della sequenza ordinata S,
tranne l'ultimo **do**

 sia y l' elemento che segue x in S

do if x=y then **return true**

return false

$T(n) = O(n \cdot \log n)$ vs $T(n) = O(n^2)$

n	n log(n)	n ²
10	33,22	100
100	664,39	10000
1000	9965,78	1000000
10000	132877,12	100000000
100000	1660964,05	10000000000
1000000	19931568,57	1000000000000
10000000	232534966,64	100000000000000
100000000	2657542475,91	10000000000000000
1000000000	29897352853,99	1000000000000000000
10000000000	332192809488,74	100000000000000000000
100000000000	3654120904376,10	10000000000000000000000
1000000000000	39863137138648,40	1000000000000000000000000
10000000000000	431850652335357,00	100000000000000000000000000
100000000000000	4650699332842310,00	10000000000000000000000000000
1000000000000000	49828921423310400,00	1000000000000000000000000000000
10000000000000000	531508495181978000,00	100000000000000000000000000000000

Misura delle prestazioni – cenni (1 di 4)

- **Tempo di CPU** relativo ad un programma: tempo effettivo durante il quale la CPU lavora su quel programma
 - Non comprende i tempi per l'accesso al disco, l'input/output,
 - Non comprende il tempo speso dalla CPU per altri programmi gestiti in contemporanea
- La **velocità o frequenza di clock** della CPU indica il numero di operazioni elementari che la CPU è in grado di eseguire in un secondo e si misura in Hertz

$$f_{\text{CLOCK}} = \# \text{ operazioni_elementari} / \text{tempo [Hertz]}$$

- Giga Hertz: $1\text{GHz} = 10^9\text{Hz} = 10^9 \text{ cicli/s}$

Misura delle prestazioni – cenni (2 di 4)

Ordini di grandezza:

- La velocità di clock del primo microprocessore della storia, l'Intel 4004, era di 740 KHz
- Le CPU dei computer moderni raggiungono i 5 GHz.

	Processore	Velocità
	Intel Core i7-8700K Migliore in assoluto	3.7 GHz
	Intel Core i5-7500 Miglior rapporto qualità prezzo	3.4 GHz
	Intel Core i7-8700 Prestazioni di altissimo livello	3.2 GHz
	Intel Core i5-7400 Ottima grafica integrata	3.0 GHz
	Intel Core i3-7100 Miglior opzione economica	3.9 GHz

Misura delle prestazioni – cenni (3 di 4)

- Tanto per quantificare:

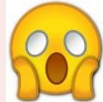
N	$N \cdot \log_2 N$	N^2	N^3	2^N
2	2	4	8	4
10	33	100	10^3	$> 10^3$
100	664	10.000	10^6	$>> 10^{25}$
1000	9.966	1.000.000	10^9	$>> 10^{250}$
10000	132.877	100.000.000	10^{12}	$>> 10^{2500}$

- Se un elaboratore esegue 1000 operazioni/sec, un algoritmo il cui tempo sia dell'ordine di 2^N richiede:

N	<i>tempo</i>
10	1 sec
20	1000 sec (17 min)
30	10^6 sec (>10giorni)
40	($>> 10$ anni)

Misura delle prestazioni – cenni (4 di 4)

- Se un elaboratore esegue $\sim 10^9$ operazioni/sec:

N	Time (N)	Time (N ²)	Time (N ³)	Time (2 ^N)
50	...	$25 \cdot 10^{-7} = 2,5 \mu\text{s}$	$125 \cdot 10^{-6} = 125 \mu\text{s}$	$> 10^6 \text{ sec} > 10 \text{ gg}$
100	$10^{-7} \text{ sec} = 0,1 \mu\text{s}$	$10^{-5} \text{ sec} = 10 \mu\text{s}$	$10^{-3} \text{ sec} = 1 \text{ ms}$	$> 10^{21} \text{ sec} \sim 10^{16} \text{ gg} > 10^{13} \text{ anni}$ 
1000	$10^{-6} \text{ sec} = 1 \mu\text{s}$	$10^{-3} \text{ sec} = 1 \text{ ms}$	1 sec	

Il problema dei duplicati: realizzazione

- Fase realizzativa: Alcune scelte, se non ben ponderate, potrebbero avere un impatto cruciale sui tempi di esecuzione
- **Implementazione dell'algoritmo `verificaDup` mediante liste:** `S` è rappresentata tramite un oggetto della classe `LinkedList` che implementa l'interfaccia `java.util.List` fornita come parte del Java Collections Framework
- Il metodo `get()` consente l'accesso agli elementi di `S` in base alla loro posizione nella lista.

Implementazione: **verificaDupList**

```
public static boolean verificaDupList (LinkedList S){  
    for (int i=0; i<S.size(); i++){  
        Object x=S.get(i);  
        for (int j=i+1; j<S.size(); j++){  
            Object y=S.get(j);  
            if (x.equals(y)) return true;  
        }  
    }  
    return false;  
}
```

Implementazione basata su ordinamento

- Utilizziamo anche la classe `java.util.Collections`, che fornisce metodi statici che operano su collezioni di oggetti
- In particolare fornisce il metodo `sort`, che si basa su una variante dell'algoritmo `mergesort`

Implementazione: **verificaDupOrdList**

```
public static boolean verificaDupOrdList (LinkedList S) {  
    Collections.sort(S);  
    for (int i=0; i<S.size()-1; i++)  
        if (S.get(i).equals(S.get(i+1))) return true;  
    return false;  
}
```

Collaudo e analisi sperimentale (1 di 7)

- L'implementazione di un algoritmo va collaudata in modo da identificare eventuali **errori implementativi**, ed analizzata sperimentalmente, possibilmente su dati di test reali
- L'analisi sperimentale delle **prestazioni** va condotta seguendo una corretta metodologia per evitare conclusioni errate o fuorvianti

Collaudo e analisi sperimentale (2 di 7)

Obiettivi dell'analisi sperimentale:

- Come raffinamento dell'analisi teorica o in sostituzione dell'analisi teorica quando questa non può essere condotta con sufficiente accuratezza
- Per effettuare un confronto più preciso tra algoritmi apparentemente simili (stimare quali sono le costanti nascoste dalla notazione asintotica)
- Per studiare le prestazioni su dati di test derivanti da applicazioni pratiche o da scenari di caso peggiore. Spesso si ottengono risultati sorprendenti la cui spiegazione consente di raffinare e migliorare l'analisi teorica
- Se un risultato sembra in contraddizione con l'analisi teorica può essere utile condurre ulteriori esperimenti

Collaudo e analisi sperimentale (3 di 7)

- **Misurazione dei tempi** (a scopo didattico in base all'orologio di sistema e basato sul clock del processore): un aspetto cruciale è la granularità delle funzioni di sistema usate per misurare i tempi. Se i tempi di esecuzione sono troppo bassi per ottenere stime significative, basta misurare il tempo totale di una serie di esecuzioni identiche dello stesso codice e dividere il tempo totale per il numero di esecuzioni
- Usiamo il metodo `java.lang.System.nanoTime()` che fornisce un valore di tipo `long` (nanosecondi) per prendere i tempi prima e dopo l'esecuzione secondo il seguente schema:

```
long tempoInizio = System.nanoTime();  
[porzione di codice da misurare]  
long tempo=System.nanoTime() - tempoInizio;
```


Collaudo e analisi sperimentale (4 di 7)

- Siamo interessati alla relazione generale esistente tra **tempo di esecuzione** e la **dimensione dei dati** da elaborare
- Si eseguono esperimenti indipendenti con molti **diversi dati** in ingresso di **diverse dimensioni**
- Si visualizzano i risultati dell'esecuzione sotto forma di **grafico cartesiano** dove la coordinata x rappresenta la dimensione n dei dati in ingresso e la coordinata y il tempo di esecuzione t
- Il grafico ottenuto consente spesso di intuire la relazione esistente tra la dimensione del problema ed il tempo di esecuzione dell'algoritmo che lo risolve

Collaudo e analisi sperimentale (5 di 7)

- Un'analisi sperimentale condotta su sequenze di numeri interi distinti generati in modo casuale ha evidenziato il vantaggio derivante dal progetto di algoritmi efficienti:
 - `verificaDupOrdList` molto più efficiente di `verificaDupList`
- I tempi di esecuzione predetti teoricamente sono rispettati? No
 - La curva dei tempi di esecuzione relativa al metodo `verificaDupList` somiglia alla funzione $c \cdot n^3$ (non a $c \cdot n^2$)
 - La curva dei tempi di esecuzione relativa al metodo `verificaDupOrdList` somiglia alla funzione $c \cdot n^2$ (non a $c \cdot n \cdot \log n$)
- **Perché ?**

Collaudo e analisi sperimentale (6 di 7)

- La contraddizione è solo apparente!
- Nell'analisi teorica abbiamo tacitamente assunto che procurarsi gli elementi in posizione i e j richiedesse tempo «costante» $O(1)$
- Controllando i dettagli dell'implementazione di `get()` ci si accorge che il metodo, avendo a disposizione solo la posizione di un elemento e non il puntatore ad esso, per raggiungere l'elemento in quella posizione è costretto a scorrere la lista dall'inizio (**esattamente i elementi**)
- Raggiungere l'elemento i -mo costa un tempo lineare **$\theta(i)$**

Collaudo e analisi sperimentale (7 di 7)

- Dunque il tempo di esecuzione di `verificaDupList` diventa proporzionale a n^3 , cioè:

$$\sum_{i=1..n} (i + \sum_{j=(i+1)..n} j) = O(n^3)$$

- Vedremo che è possibile migliorare l'implementazione (tempo di esecuzione quadratico $O(n^2)$) !
- Discorso analogo vale per il metodo `verificaDupOrdList`.

Messa a punto e ingegnerizzazione

- Richiede in particolare di decidere l'organizzazione e la modalità di accesso ai dati
- In riferimento al nostro esempio, dove la sequenza S è rappresentata mediante un oggetto `LinkedList`, l'uso incauto del metodo `get()` ha reso le implementazioni inefficienti
- Eliminare questa fonte di inefficienza: convertire la lista in array!

Implementazione: **verificaDupArray**

```
public static boolean verificaDupArray (List S) {  
    Object[] T = S.toArray();  
    for (int i=0; i<T.length(); i++) {  
        Object x=T[i];  
        for (int j=i+1; j<T.length; j++) {  
            Object y=T[j];  
            if (x.equals(y)) return true;  
        }  
    }  
    return false;  
}
```

Implementazione: **verificaDupOrdArray**

```
public static boolean verificaDupOrdArray (List S) {  
    Object[] T = S.toArray();  
    Arrays.sort(T);  
    for (int i=0; i<T.length(); i++) {  
        if (T[i].equals(T[i+1])) return true;  
    }  
    return false;  
}
```

- I tempi di esecuzione in questo caso sono perfettamente allineati con la predizione teorica!

Brainstorming

- Cosa influisce sull'efficienza di un algoritmo?
 - L'idea algoritmica (ovviamente!)
 - L'organizzazione dei dati
 - Ad es. ordinare preliminarmente un array di elementi consente di applicare l'algoritmo di ricerca binaria
 - L'uso di strutture dati avanzate può migliorare il tempo di esecuzione
 - Ad es. l'uso di un heap binario consente di implementare un algoritmo di selezione più intelligente per l'ordinamento di un array



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica



Domande?

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM