Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze
dell'Informazione e Matematica

Università degli Studi dell'Aquila

Modulo di Laboratorio di Algoritmi e Strutture Dati

Algoritmi e loro implementazione in Java:
Introduzione

Algoritmo

- Informalmente, un **algoritmo** è un **procedimento effettivo** che consente di risolvere un **problema** (ovvero di ottenere una risposta ad un determinato quesito) eseguendo, in un determinato ordine, un insieme finito di passi semplici (**azioni**), scelti tra un insieme (solitamente) finito di possibili azioni
- 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 devo essere organizzati e strutturati in modo tale che la procedura che li elabora sia “efficiente”

⇒ Il concetto di algoritmo è inscindibile da quello di dato

- Ci focalizziamo su algoritmi pensati per risolvere problemi di calcolo la cui soluzione può essere delegata alla CPU di un sistema di elaborazione automatica.

Caratteristiche di un algoritmo

- La sequenza di istruzioni deve essere finita (**finitezza**);
- La procedura deve portare ad un risultato corretto (**effettività**);
- Le istruzioni devono essere eseguibili materialmente (**realizzabilità**);
- Le istruzioni devono essere espresse in modo non ambiguo (**non ambiguità**)
- Ogni algoritmo è caratterizzato da una **complessità temporale e spaziale** rispetto alle dimensioni dei dati di ingresso.

Ciclo di sviluppo di codice algoritmico: cenni

- Lo sviluppo di software robusto ed efficiente per la soluzione di problemi di calcolo richiede (tra le altre cose):
  - Creatività
  - Capacità di astrazione
  - Familiarità di strumenti matematici
  - Padronanza del linguaggio di programmazione
- Schema semplificato a due fasi che si avvicendano in un processo ciclico:
  - **Fase progettuale**
  - **Fase realizzativa**

Fase progettuale

1. 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:
Esempi di definizione dei requisiti di un problema

Problema di ordinamento:

- **Input:** un insieme di elementi qualsiasi $A=\{a_1, . . . , a_n\}$ 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 $h \leq k$ ($h, k=1, 2, . . . ,n$)

Problema di ricerca:

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

Fase progettuale

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

$\Rightarrow$ 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. Si progetta un **algoritmo risolutivo**, verificandone formalmente la **correttezza** e stimandone le **prestazioni teoriche**
   - Per uno stesso problema algoritmico esistono più algoritmi risolutivi
   - L'obiettivo è trovare l'algoritmo che faccia un uso ottimale delle risorse di calcolo disponibili (**tempo di esecuzione** ed **occupazione di memoria**)

$\Rightarrow$ In grossi progetti sw è fondamentale <u>stimare le prestazioni già a livello progettuale</u>. 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

4. Qualora la verifica della correttezza rilevi problemi o la stima delle prestazioni risulti poco soddisfacente si torna al passo 3 (se non al passo 2…)

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

- Formulato come un problema di decisione
- **Input**: una sequenza S di elementi qualsiasi $S=\{s_1, . . . , s_n\}$,
- **Output**: *true* se esiste in S una coppia di elementi duplicati (cioè esiste in S una coppia di indici distinti $i, j \in \{1, . . . ,n\}$ tale che $s_i = s_j$), *false* altrimenti

```
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
```

- **Analisi della 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.

Il problema dei duplicati

- **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 ordinare 10 elementi e per ordinarne 1 milione è lo stesso?*

Il problema dei duplicati

- Usiamo la **notazione asintotica** per esprimere le delimitazioni inferiori e superiori alla complessità di un problema rispetto ad una data risorsa di calcolo, ossia:
- 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) <u>per qualsiasi algoritmo</u> 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 <u>uno specifico algoritmo</u> che risolve una generica istanza del problema dato.

Il problema dei duplicati

Analisi del tempo di esecuzione di `verificaDup` per una generica istanza di dimensione n (per $n \to \infty$):

- informalmente, per valutare l'ordine di grandezza "O(·)" 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") per istanze più sfavorevoli (**caso peggiore**)

Difficoltà intrinseca del problema:

- delimitazione inferiore banale di ogni algoritmo: $\Omega(n)$ (almeno la lettura dei dati in ingresso)

$\Rightarrow$ Esistono algoritmi più efficienti di **`verificaDup`**?

Il problema dei duplicati

- 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
- Idea nuovo algoritmo:
  - Ordinare la sequenza ($\theta(n \cdot \log n)$, 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

```
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
```

Ordini di grandezza

- La **velocità o frequenza di clock della CPU** indica il numero di operazioni elementari che la CPU è in grado di eseguire nell'arco di un secondo:

$$f_{CLOCK} = Numero\_operazioni\_elementari \,/\, tempo \; [Hertz]$$

- La velocità di clock del primo microprocessore della storia, l'Intel 4004, era di 740 KHz
- Le CPU dei computer moderni raggiungono quasi i 4 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

Ordini di grandezza

- Tanto per quantificare:

N	$N*\log_2 N$	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	13.288	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	*tempo*
10	1 sec
20	1000 sec (17 min)
30	10^6 sec (>10giorni)
40	(>>10 anni)

Ordini di grandezza

- Se un elaboratore più moderno esegue ~10^9 operazioni/sec

N	Time (N)	Time (N^2)	Time (N^3)	Time (2^N)
50	…	$25 \cdot 10^{-7}$= 2,5 μs	$125 \cdot 10^{-6}$ = 125 μs	> 10^6 sec ~ 10 gg
100	10^{-7} sec= 0,1 μs	10^{-5} sec= 10 μs	10^{-3} sec = 1 ms	> 10^{21} sec ~ 10^{13} gg > 320000 anni
1000	10^{-6} sec = 1 μs	10^{-3} sec = 1 ms	1 sec	> 10^{2491} sec

Il problema dei duplicati

- **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 (che vedremo in seguito…)
- Il metodo `get` consente l'accesso agli elementi di S in base alla loro posizione nella lista.

Il problema dei duplicati

```
public static boolean verificaDupList (List 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;
}
```

Il problema dei duplicati

- 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`

```
public static boolean verificaDupOrdList (List 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

- L’implementazione di un algoritmo va collaudata in modo da identificare eventuali errori implementativi, ed analizzata sperimentalmente, possibilmente su dati di test reali
  - Durante l’analisi sperimentale spesso si ottengono risultati sorprendenti la cui spiegazione consente di raffinare i modelli di calcolo o l’analisi teorica stessa aprendo la strada a possibili miglioramenti
  - I dati di test reali possono presentare caratteristiche che agevolano o mettono in difficoltà l’algoritmo, che potrebbe essere migliorato in contesti specifici
  - L’analisi sperimentale consente di capire quali sono le costanti nascoste dalla notazione asintotica ottenendo un confronto più preciso tra algoritmi apparentemente simili

Il problema dei duplicati

- **Collaudo e analisi sperimentale** (**duplicati**):
- L'analisi sperimentale (per dettagli rif. libro di testo) condotta su sequenze di numeri interi distinti generati in modo casuale evidenzia 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 $c*n^3$ (non a $c*n^2$)
  - La curva dei tempi di esecuzione relativa al metodo `verificaDupOrdpList` somiglia alla funzione $c*n^2$ (non a $c*n*\log n$)
- **Perché ?**

Il problema dei duplicati

- La contraddizione è solo apparente!
- Nell'analisi teorica abbiamo tacitamente assunto che procurarsi gli elementi in posizione i e j richiedesse tempo 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:
- Raggiungere l'elemento i-mo costa $\theta(i)$

Il problema dei duplicati

- Dunque il tempo di esecuzione di `verificaDupList` diventa proporzionale a:

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

- È semplice mostrare che anche la delimitazione inferiore è $\Omega(n^3)$, dunque `verificaDupList` ha tempo di esecuzione $\theta(n^3)$.
- Vedremo che è possibile migliorare l'implementazione (tempo di esecuzione $O(n^2)$) !
- Discorso analogo vale per il metodo `verificaDupOrdList`.

Collaudo e analisi sperimentale

Metodologie di analisi sperimentale (cenni)

- L’analisi sperimentale delle prestazioni va condotta seguendo una corretta metodologia per evitare conclusioni errate o fuorvianti

Obiettivi:

- Come raffinamento dell’analisi teorica o in sostituzione dell’analisi teorica quando questa non può essere condotta con sufficiente accuratezza
- Per stimare le costanti moltiplicative ignorate
- Per studiare le prestazioni su dati di test derivanti da applicazioni pratiche o da scenari di caso peggiore
- Se un risultato sembra in contraddizione con l’analisi teorica può essere utile condurre ulteriori esperimenti

Collaudo e analisi sperimentale

Le analisi sperimentali: l'impianto sperimentale è caratterizzato da molteplici aspetti, la cui conoscenza è fondamentale per interpretare i risultati in modo corretto:

- **Piattaforma**: come piattaforma di sperimentazione per analizzare gli algoritmi utilizzeremo il *Java RunTime Environment*
- **Misure di qualità del codice**: ci concentreremo sull'uso delle risorse di calcolo (tralasceremo la qualità della soluzione approssimata)

Collaudo e analisi sperimentale

- **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:

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

Collaudo e analisi sperimentale

- 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

- **Dati di test**: è opportuno usare:
  - insiemi di test il più possibile generali
  - Istanze realistiche per le specifiche applicazioni
- **Riproducibilità dei risultati**: è importante documentare il lavoro in modo preciso in modo da consentire la riproduzione dei risultati

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 List, l'uso incauto del metodo get ha reso le implementazioni inefficienti
- Eliminare questa fonte di inefficienza: convertire la lista in **array**!

Il problema dei duplicati

```
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;
}
```

Il problema dei duplicati

```
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!