Introduzione agli Algoritmi

Prof. Emanuela Fachini

- Contenuto e motivazioni del corso
- Qualche anticipazione
- organizzazione ed esami

Pagina del corso: http://twiki.di.uniroma1.it/twiki/view/Intro_algo/AD/WebHome

Cos'è un algoritmo?

Gli algoritmi sono le idee sulle quali si basano i programmi.

Gli algoritmi risolvono problemi generali che devono essere ben specificati.

Si devono specificare le istanze del problema e qual'è l'output atteso per ogni istanza.

Da dove viene la parola?

La traduzione latina, risalente al XII secolo, di un libro di Muhammad ibn Mūsa 'l-Khwārizmī era intitolata "l-Khwārizmī sui numeri indiani", che fu interpretato come "algoritmi sui numeri indiani".

Il matematico persiano, Muhammad ibn Mūsa 'l-Khwārizmī, nato nel 780 circa e morto nel 850, è famoso per il libro "Kit āb al-djabr wa 'l-muq ābala " il primo libro che tratta in modo sistematico le equazioni lineari e di secondo grado e dal cui titolo è nato anche il termine algebra (al-djabr)

Esempio di problema

Il problema della ricerca di un elemento x in una lista ordinata di elementi.

E' ben specificato?

NO

La lista contiene duplicati?

Quale output è richiesto?

Esempio di problema ben specificato

Il problema di determinare la posizione in una lista ordinata e priva di duplicati di un elemento x presente nella lista. Se invece x non è presente nella lista si vuole -1 in output.

Un'istanza del problema (che è anche l'input per l'algoritmo) è una coppia formata da una lista e da un elemento da cercare.

L'output è la posizione di x nella lista, se presente, -1 se assente.

Esempio di algoritmo

L'algoritmo della ricerca binaria.

Idea: confrontiamo l'elemento cercato con quello centrale nella lista, se coincidono allora diamo in output l'indice di quell'elemento, altrimenti ripetiamo il procedimento sulla metà sinistra della lista, se l'elemento cercato è minore del centrale, o sulla metà destra altrimenti.
Il procedimento si ferma quando si trova x, oppure, se x non è presente, quando la ricerca finisce in un intervallo vuoto.

È corretto?

Prova induttiva sul numero degli elementi.
Base. Se la lista contiene un solo elemento che coincide con quello cercato allora il risultato sarà la sua posizione nella lista cioè 0, altrimenti -1.
Passo induttivo. Supponiamo che il nostro algoritmo lavori correttamente su un numero di elementi minore di n. Ora applichiamolo a una lista di n elementi. Se quello cercato è minore del centrale allora la ricerca si dirige nella metà sinistra, se è maggiore in quella destra. Le due metà hanno meno di n elementi e quindi la ricerca si concluderà correttamente

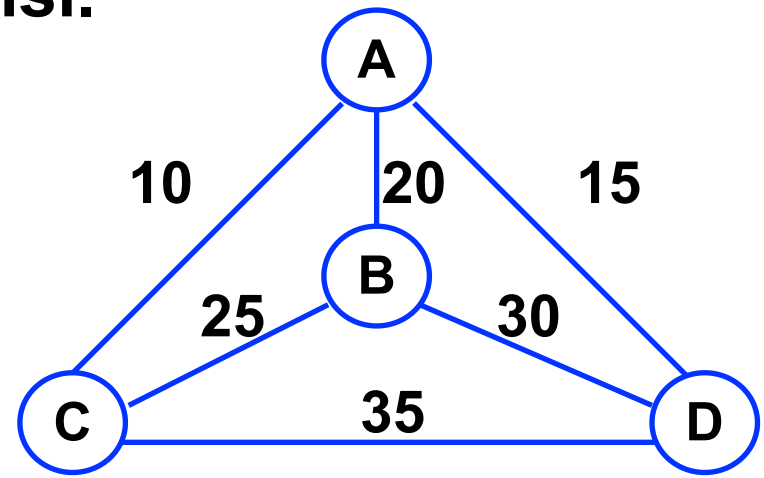
Correttezza

Dimostrare la correttezza di un algoritmo non è sempre facile, ma è essenziale!

Problema del Commesso Viaggiatore

Dato un insieme di città con le distanze delle strade di collegamento tra ogni coppia di città, il problema consiste nel determinare il giro più corto che attraversa tutte le città una sola volta, tornando al punto di partenza. Le vie di collegamento sono percorribili nei due sensi.

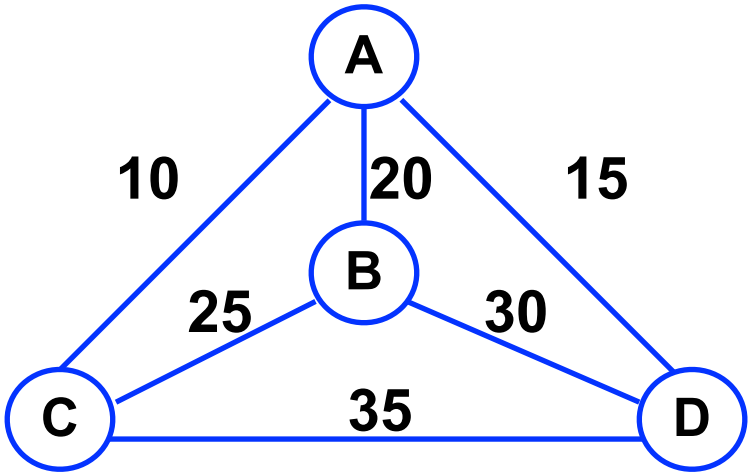
Istanza:



Un algoritmo

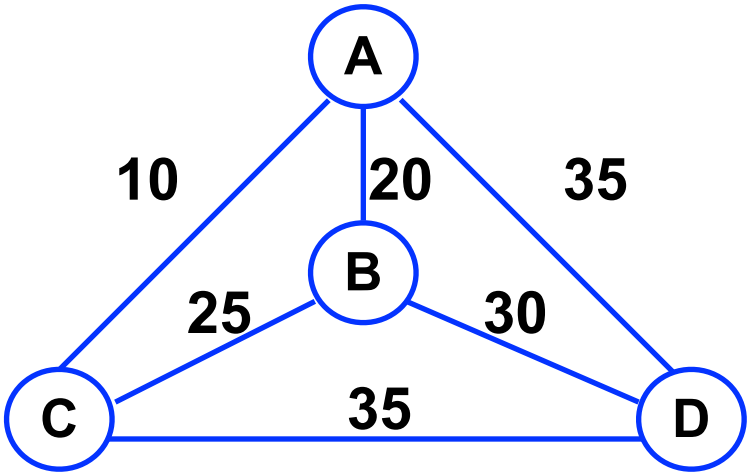
Partendo da una qualunque città, prendiamo la più vicina tra quelle ad essa collegata e non ancora visitata, fino a tornare alla città di partenza.

In questo caso, partendo da A, prendendo C, poi B, D e A, si crea un percorso lungo 80. Si può verificare che ogni altro percorso ha una lunghezza maggiore.



È corretto?

In questo altro caso, partendo da A, prendiamo C, poi B, D e A, un percorso lungo 100, mentre il percorso A,C,D,B,A è lungo 95.



Eppure il nostro criterio sembrava molto ragionevole!
Dobbiamo affidarci al nostro intuito ma anche diffidare di esso!

Conclusione

La prima cosa da verificare, una volta concepito un algoritmo è la sua correttezza, cioè controllare se per ogni istanza del problema l'algoritmo dà la risposta giusta.

Un altro algoritmo per il problema del commesso viaggiatore

Considera tutti i possibili giri che partono e tornano in una città, calcolane la lunghezza e scegli quello che ha lunghezza minima.

È corretto?

Banalmente sì perchè si basa su una ricerca esaustiva tra tutte le possibili soluzioni. Si calcola la distanza per ogni giro, dato da una permutazione della lista delle città, e si conserva quello di distanza minima corrente.

Analisi dell'algoritmo

È efficiente?

Quanti sono i giri che si devono considerare tra n città?
Ogni giro corrisponde a una permutazione delle città.
Quante sono le permutazioni su n elementi?
Sono n!
E allora se per esempio ci sono 20 città, 20! = 2.432.902.008.176.640.000 che è circa 10^{18}.
Ammettendo che ogni passo di calcolo possa essere eseguito in un microsecondo cioè in 10^{-6} secondi, l'algoritmo terminerebbe in più di 3000 anni!

Conclusione

Si deve verificare, una volta concepito un algoritmo, il tempo di calcolo e la quantità di memoria necessari per il calcolo.

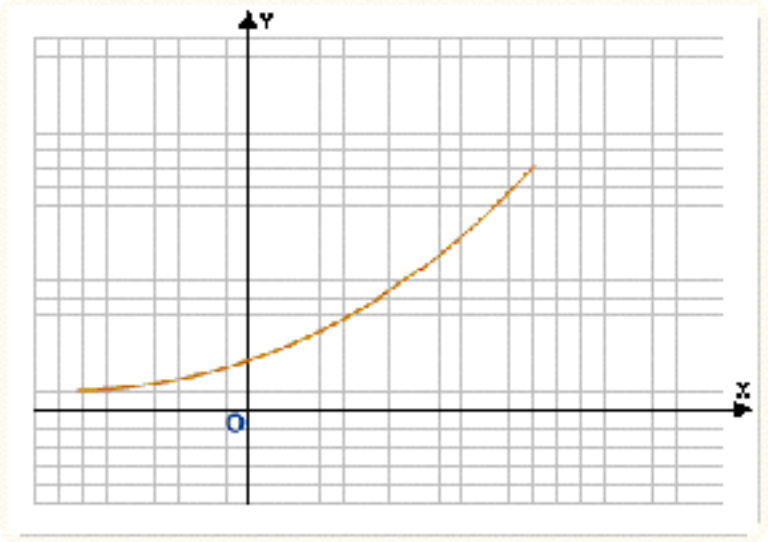
Sono le misure dell'efficienza di un algoritmo.

Ma come misurarli?

Analisi degli algoritmi: come?

- **Approccio sperimentale**

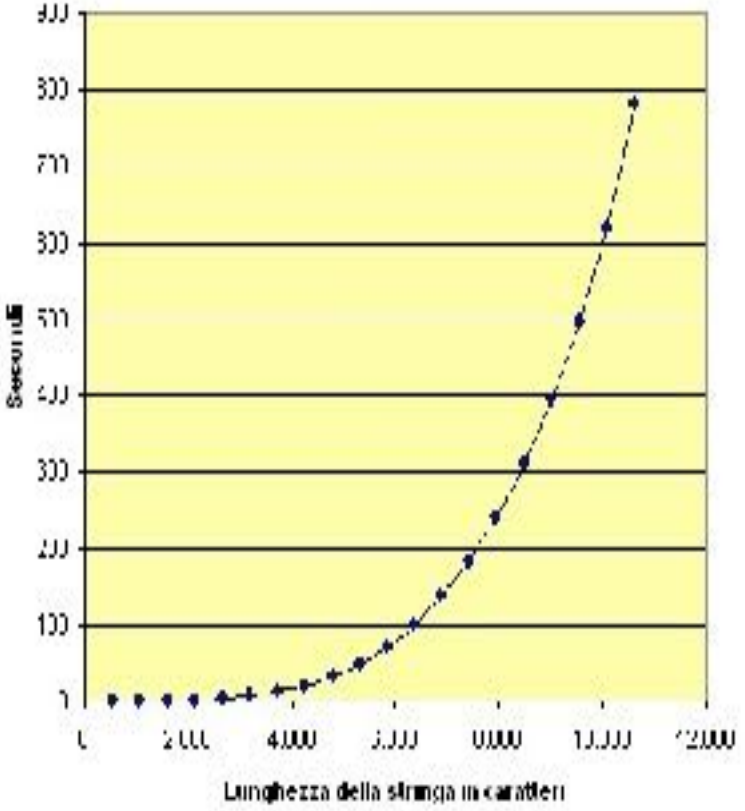
- **Approccio teorico**



Approccio sperimentale

Esperimenti.

Si fa girare un programma che implementa l'algoritmo su input di dimensione crescente e se ne calcolano i tempi di esecuzione. Da questi dati si possono fare previsioni sul comportamento dell'algoritmo su dati di dimensione maggiore che possono essere controllate con nuovi esperimenti.



Approccio sperimentale: limiti

Limiti:

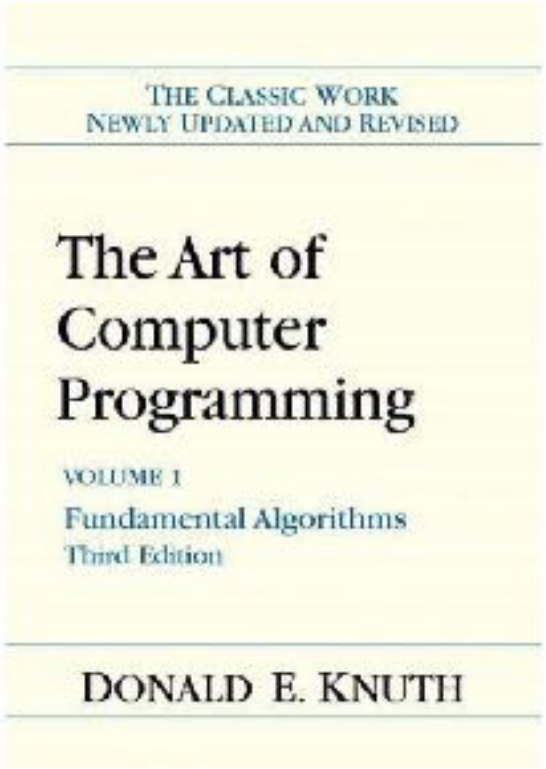
- **Si deve implementare l'algoritmo**

- **Non sempre i risultati sono indicativi del tempo di calcolo su input non compresi nell'esperimento**

- **Il contronto tra algoritmi deve basarsi sullo stesso ambiente hardware/software (processore, tipo e organizzazione memoria, Sistema Operativo, il compilatore,…)**

Metodo teorico

Tempo totale di calcolo di un algoritmo: somma dei tempi di ogni operazione eseguita



Analisi di un algoritmo : si calcola un'approssimazione sul numero di operazioni eseguite in tempo costante.
Si definisce una funzione che associa tale valore approssimato alla dimensione dell'input.

Metodo teorico: vantaggi

- usa una descrizione ad alto livello dell'algoritmo

- dà una risposta per ogni possibile input

- permette di valutare l'algoritmo indipendentemente dall'ambiente hard/software

Esempio 1

```
if score >= 90:
    print('A')
else:
    if score >=80:
        print('B')
```

Tempo di calcolo di un algoritmo:
La somma dei tempi di esecuzione delle operazioni di costo costante che l'algoritmo esegue.

Tempo di calcolo:
il primo if prende un tempo costante, diciamo a
il secondo ancora una costante, diciamo c,
print prende tempo costante b,
mentre il confronto ha un tempo costante d
allora il tempo totale è al più 2d + a + 2b + c.
Questo dice che questo frammento di programma ha un tempo di esecuzione costante.

Esempio 2

```
1. wordlist = ['cat','dog','rabbit']
2. letterlist = [ ]
3. for aword in wordlist:
4.     for aletter in aword:
5.         letterlist.append(aletter)
6. print(letterlist)
```

tempo di calcolo:
Le linee 1,2 e 6 sono eseguite in tempo costante **a**, **b** e **c**.
La linea 5 è eseguita in tempo costante, **d**.

Il ciclo for della linea 3 è eseguito tante volte quante sono le parole in wordlist, nell'esempio 3 volte.
Ma se vogliamo una previsione su una lista qualunque?
Allora chiamiamo **n** il numero delle parole nella lista, e possiamo dire che che è eseguito **n** volte.

Esempio 2 - cont.

```
1. wordlist = ['cat','dog','rabbit']
2. letterlist = [ ]
3. for aword in wordlist:
4.     for aletter in aword:
5.         letterlist.append(aletter)
6. print(letterlist)
```

Il secondo ciclo (linea 4) è eseguito tante volte quant'è la lunghezza della parola scelta nella lista. Se supponiamo che la più lunga parola nella lista è lunga m, possiamo dire che il tempo di esecuzione su un input formato da una lista di n parole, di lunghezza al più m, è minore o uguale a a+b+c + n*m*d.

Esempio di analisi di un algoritmo di ordinamento

Specificazione del problema

Problema dell'ordinamento:

data una lista di n elementi A= <a1,a2,...,an> presi da un insieme totalmente ordinato si tratta di produrre una lista ordinata degli n elementi, secondo uno dei possibili versi (crescente o descrescente) dell'ordinamento.

Se gli elementi sono numeri naturali allora un' instanza del problema è una sequenza di numeri A = <5,8,7,6,10,43,67> e con l'ordinamento naturale l'output, relativo ad essa, è:

Output: A = <5,6,7,8,10,43,67> crescente o
A = <67,43,10,8,7,6,5> decrescente

Specificazione del problema: secondo esempio

Se gli elementi sono stringhe di caratteri allora un' istanza del problema è una sequenza di stringhe (o parole): B=<pesche, mele, pere, ciliege,albicocche>

Con l'ordinamento crescente lessicografico (alfabetico) l'output, relativo all'istanza è:

B = <albicocche,ciliegie,mele,pere,pesche, uva>

Con l'ordinamento crescente quasi lessicografico (per lunghezza e a parità di lunghezza in alfabetico) l'output, relativo alla stessa istanza è:

B = < uva,mele,pere,pesche,ciliegie, albicocche>

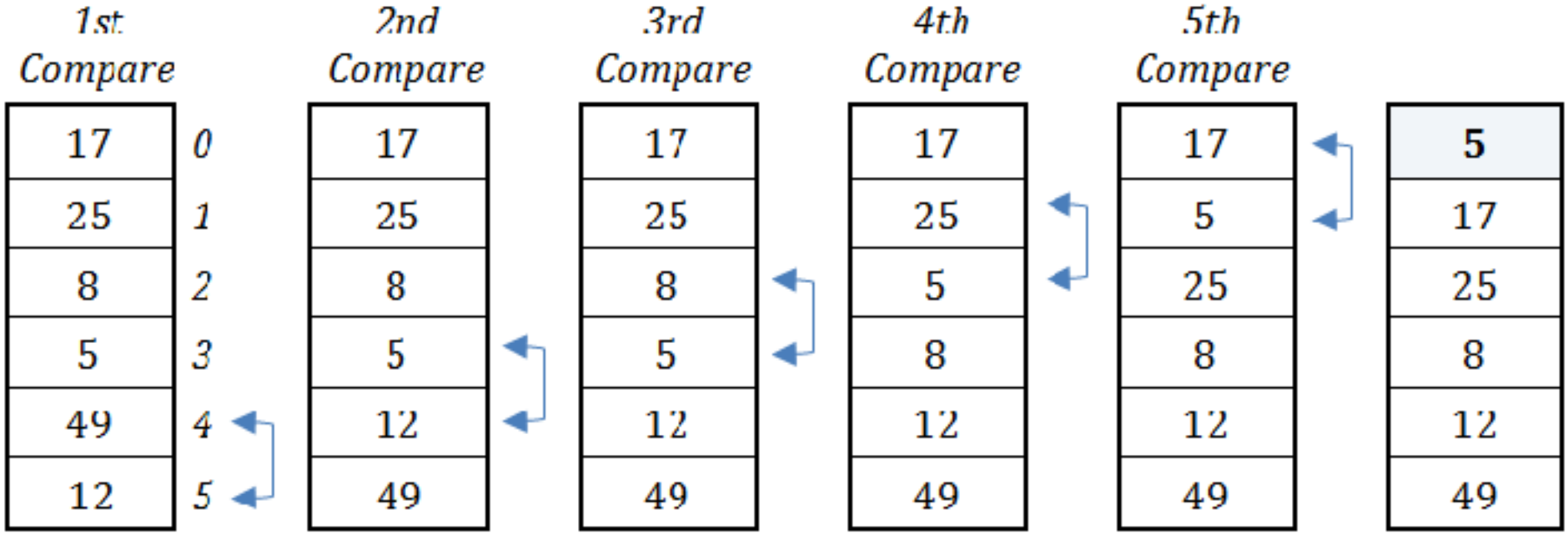
Un algoritmo di ordinamento: BubbleSort

Abbiamo una lista di n elementi da ordinare in ordine crescente.

Si considerano tutte le coppie di elementi consecutivi nella lista a partire da destra (ma si potrebbe anche partire da sinistra). Si confrontano i due elementi della coppia e si scambiano di posto se il secondo è minore del primo, in modo che la coppia risulti ordinata. L'algoritmo continua a eseguire questi passaggi per tutta la lista finché non vengono più eseguiti scambi, situazione che indica che la lista è ordinata.

Esempio di esecuzione del BubbleSort

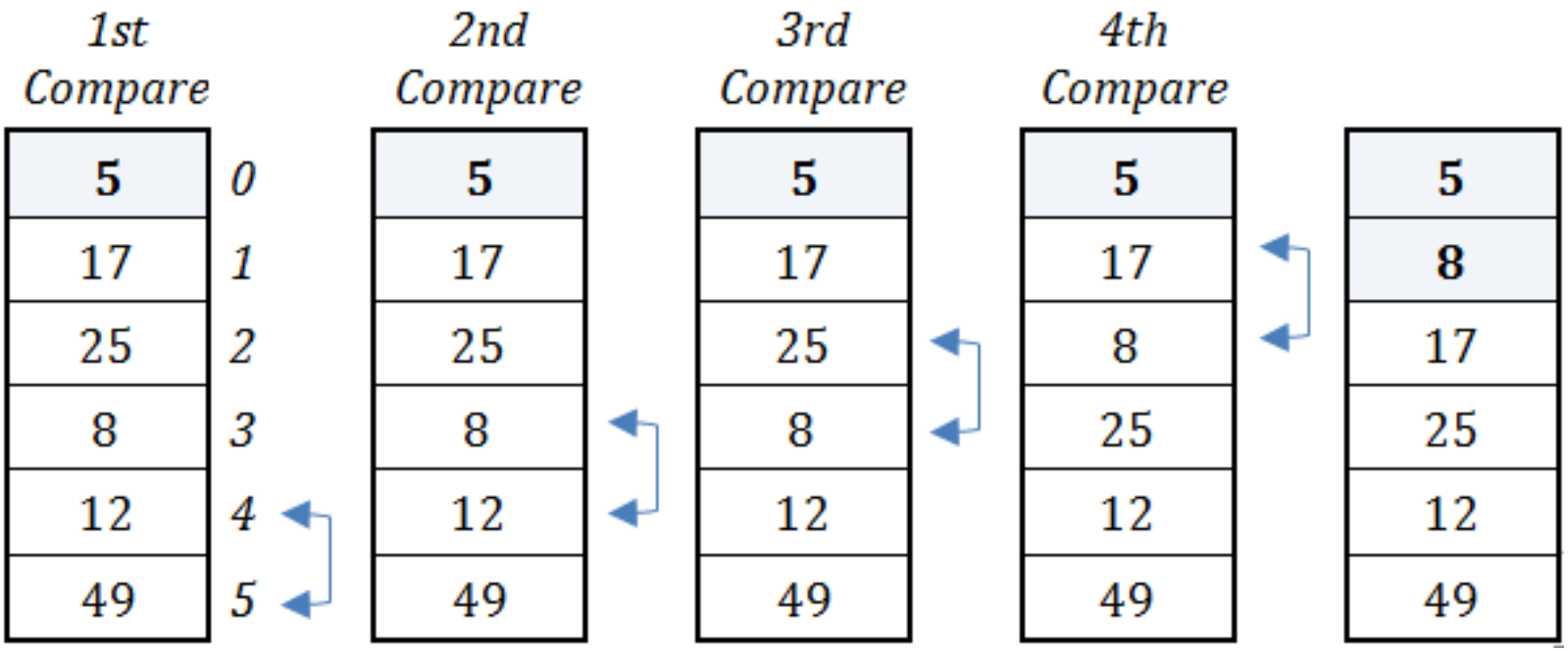
Fase 1: Primo scorrimento degli elementi



Esempio preso da:
http://www.bouraspage.com/repository/algorithmic-thinking/the-bubble-sort-algorithm-sorting-one-dimensional-arrays-with-numeric-values

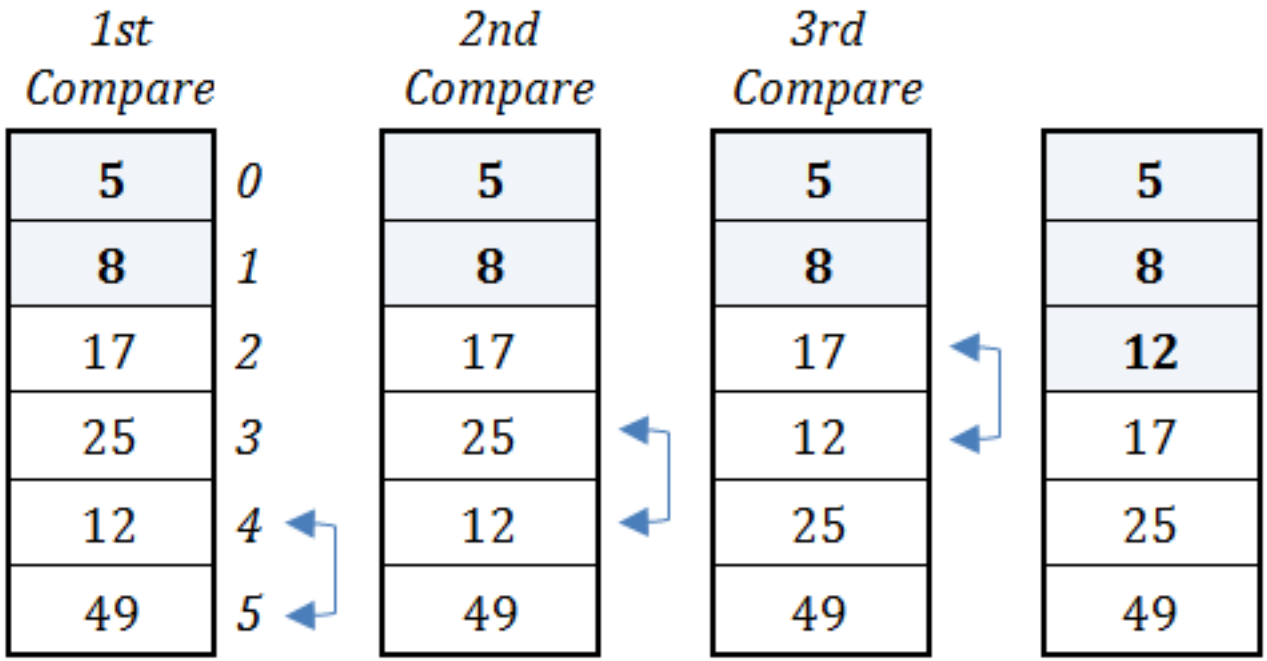
Esempio di esecuzione del BubbleSort

Fase 2 : secondo scorrimento degli elementi



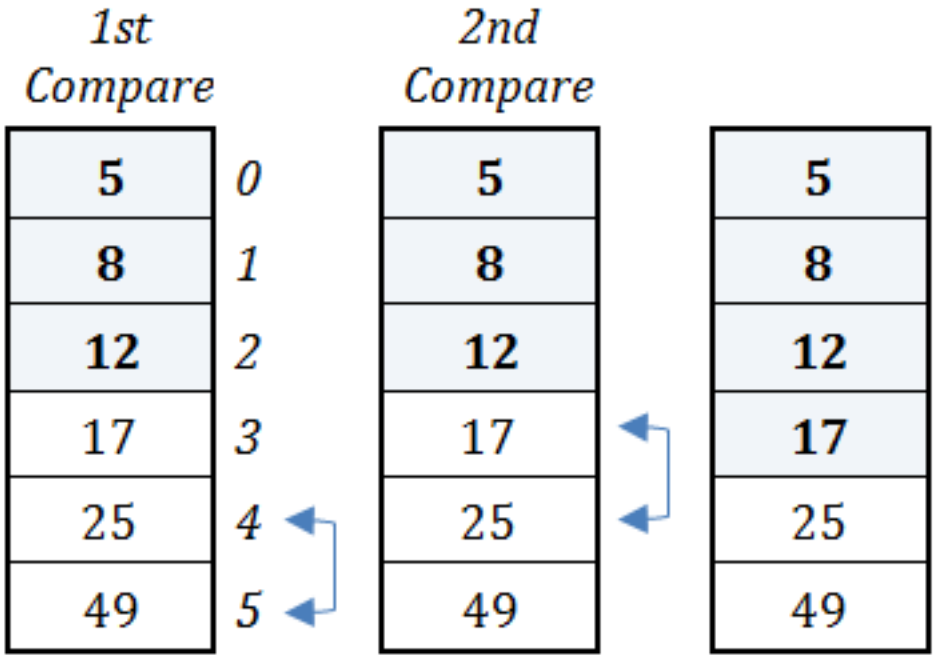
Esempio di esecuzione del BubbleSort

Fase 3: terzo scorrimento degli elementi



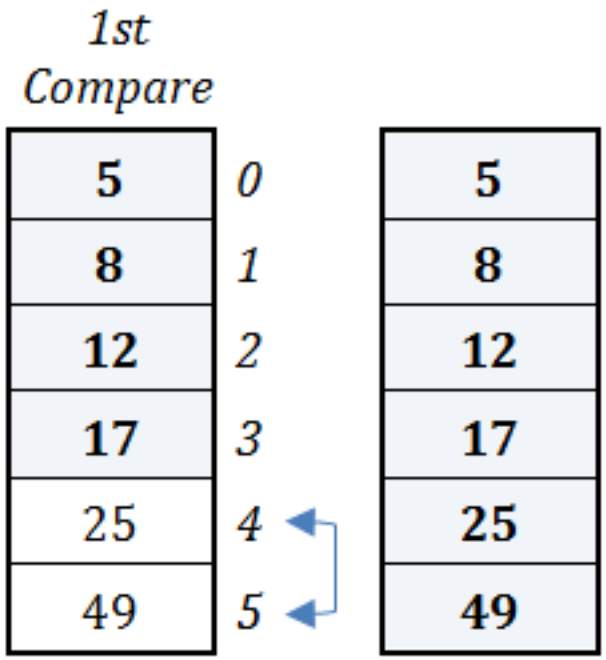
Esempio di esecuzione del BubbleSort

Fase 4: quarto scorrimento degli elementi



Esempio di esecuzione del BubbleSort

Fase 5: quinto scorrimento degli elementi



Correttezza del BubbleSort

Prova induttiva sul numero degli scorrimenti. Proviamo che dopo n scorrimenti, n elementi sono ordinati e nelle prime posizioni della lista.
Base. Dopo un unico scorrimento il minimo della lista è nella prima posizione. Infatti il minimo verrà scambiato con tutti gli elementi con cui viene confrontato.
Passo induttivo. Dopo m scorrimenti i primi m elementi sono ordinati nelle prime m posizioni della lista e sono minori dei rimanenti elementi nelle posizioni da m+1 a n. Un ulteriore scorrimento porterà il minimo tra gli elementi di indice m+1 fino a n , nella posizione m+1.

Ipotesi per l' analisi del tempo di esecuzione

Prima di procedere all'analisi vera e propria dell'algoritmo dobbiamo stabilire quali operazioni consideriamo eseguite in tempo costante.

In prima istanza consideriamo eseguiti in tempo costante:

i confronti tra numeri

tutti gli assegnamenti

le operazioni aritmetiche (incrementi, somme, prodotti tra numeri, etc.)

Ipotesi per l' analisi del tempo di esecuzione

In prima istanza consideriamo eseguiti in tempo costante:

i controlli sul flusso di esecuzione (if, while, for, etc.)

anche espressi in italiano, come la frase "ripeti, finché non vengono più eseguiti scambi", purché ragionevoli.

Conteggio del numero di confronti nel BubbleSort

Si considerano tutte le coppie di elementi consecutivi nella lista a partire da destra. Si confrontano i due elementi della coppia e si scambiano di posto se il secondo è minore del primo, in modo che la coppia risulti ordinata.

L'algoritmo continua a eseguire questi passaggi per tutta la lista finché non vengono più eseguiti scambi, situazione che indica che la lista è ordinata.

Si eseguono n - 1 confronti ogni fase, se non si tiene conto del fatto che dopo m fasi i primi m elementi sono già ordinati e non è necessario fare altri confronti su di essi.

Così il tempo di esecuzione è C(n-1)n

Conteggio del numero di confronti nel BubbleSort migliorato

Si considerano tutte le coppie di elementi consecutivi nella lista a partire da destra. Nella prima fase si confrontano i due elementi della coppia e si scambiano di posto se il secondo è minore del primo, in modo che la coppia risulti ordinata. L'algoritmo continua a eseguire questi passaggi. Nella fase i-sima si confrontano gli elementi dall'ultimo all'(i+1)-simo. Le fasi si ripetono finché non vengono più eseguiti scambi, situazione che indica che la lista è ordinata.

Si eseguono n - 1 confronti, nella prima fase,

n-2 nella seconda,

n-3 nella terza, … ,

fino all'(n-1)-sima fase nella quale si esegue un confronto.

Conteggio del numero di operazioni eseguite in tempo costante nel BubbleSort

Per ogni confronto può essere eseguito uno scambio. Quindi possiamo concludere che il numero delle operazioni eseguite per una lista di n elementi è superiormente limitato da

$$Cn(n-1)/2$$

dove C è una costante che dà conto del tempo di esecuzione di ogni confronto e ogni scambio.

Prima conclusione esempio

Abbiamo ottenuto una stima, più precisamente una limitazione superiore, del tempo di calcolo di un algoritmo senza bisogno nemmeno di descriverlo in pseudocodice.

Lo pseudocodice diventa utile per descrivere alcuni dettagli dell'algoritmo, quindi conviene in genere descrivere l'idea in italiano, convincersi della sua correttezza, fare l'analisi del tempo di calcolo e poi entrare in maggior dettaglio usando lo pseudocodice.

Anche la stima può raffinarsi di conseguenza.

Seconda conclusione esempio

Abbiamo ottenuto una stima, più precisamente una limitazione superiore, del tempo di calcolo di un algoritmo sotto l'ipotesi che i confronti, gli scambi e i costrutti necessari per eseguire tutte le fasi (un while o equivalente costrutto per gestire un ciclo) sono eseguiti in tempo costante.

In altre parole abbiamo implicitamente adottato un modello di calcolo.

Algoritmi e programmi

	programmi	algoritmi
espressi in	**linguaggio di programmazione**	**linguaggio naturale o pseudocodice**
eseguiti su (girano)	**un calcolatore**	**un modello di calcolo**

Modello di calcolo

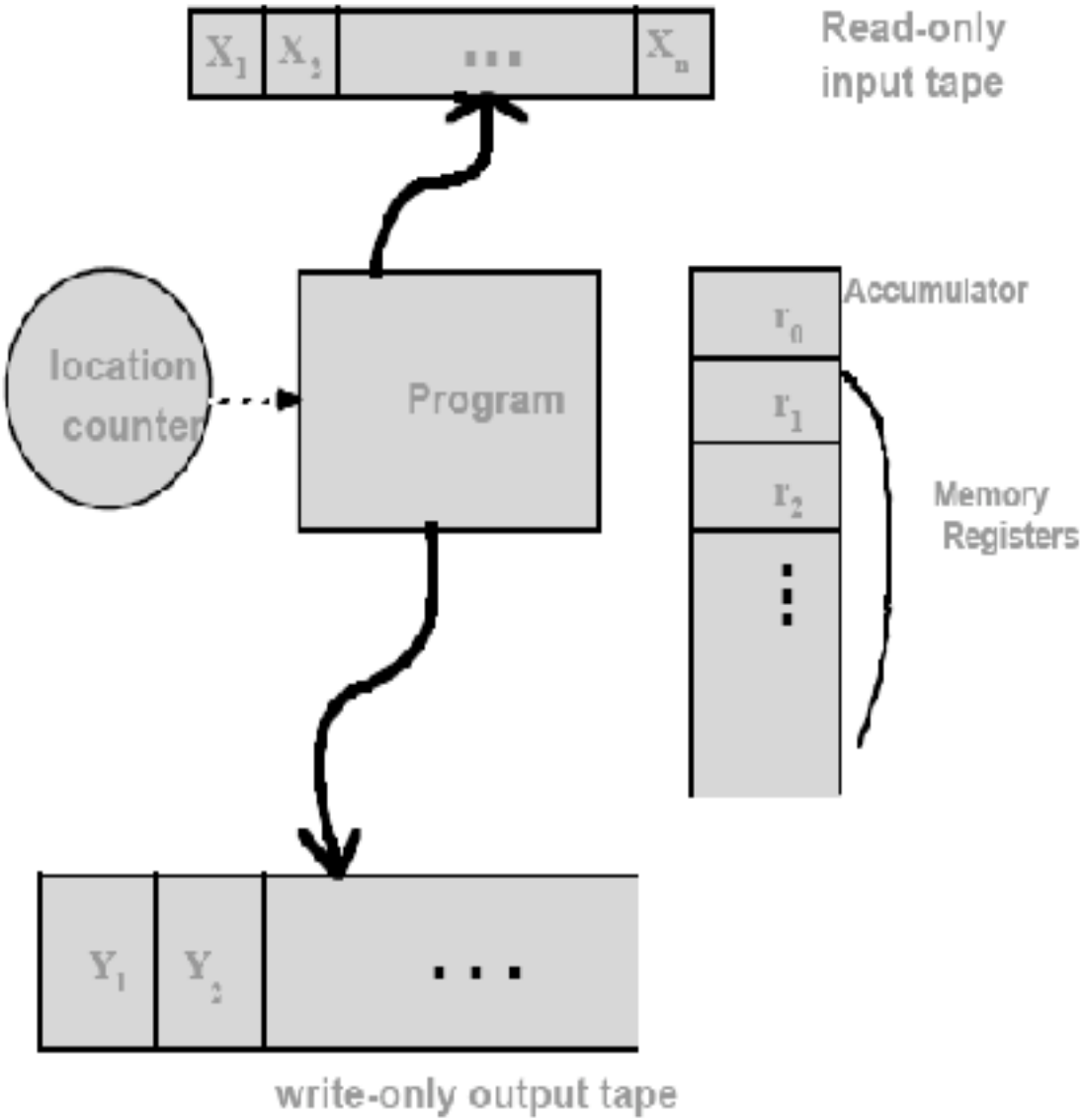
Un modello di calcolo è definito precisando un modello architetturale e quali operazioni sono permesse per costruire un algoritmo e con quale costo.

Il costo è in termini di tempo di calcolo, ma anche di spazio di memoria.

Un modello serve per semplificare e astrarre dalle specificità di un'architettura di calcolatore.

Utilzzeremo il modello RAM: Random Access Machine

RAM: architettura



La RAM dispone di un unico processore e di un numero finito di registri.

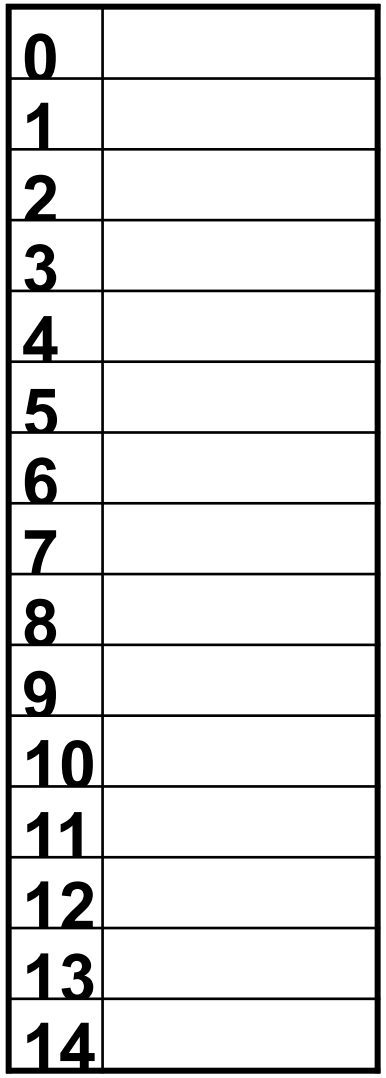
Ogni registro può memorizzare una parola, i dati in memoria vengono caricati (load) nei registri e da questi i dati vengono memorizzati (store) nella memoria RAM.

RAM: la memoria

In una Random Access Machine la memoria è una Random Access Memory:

un array di celle di memoria, ciascuna accessibile via l'indirizzo e in grado di contenere una word (*parola*), cioè una sequenza di un numero finito di bit (nella realtà 32 o 64).

Anche l'indirizzo è una word.



Random Access Memory

Ipotesi sul modello RAM

- Ogni registro può contenere un intero
- Il programma non modifica sé stesso
- Ogni accesso alla memoria prende tempo costante
- Le istruzioni di base sono semplici:
  a) Addizione, sottrazione
  b) Moltiplicazione, divisione e istruzioni di controllo (while, if-then, etc.)
- Ogni istruzione di base è eseguita in tempo costante

Ipotesi sul modello RAM - 2

Cicli e chiamate di funzioni non sono ingenerale operazioni eseguibili in tempo costante, ma hanno un tempo di esecuzione che dipende dalla dimensione dell'input (numero degli elementi, come nel caso dell'ordinamento, lunghezza di una stringa, etc.).

Algoritmi di ordinamento

Studieremo vari algoritmi di ordinamento,

perché ordinare i dati è fondamentale nella maggioranza delle applicazioni.
Si stima che circa il 25% del tempo di calcolo sia utilizzato per l'ordinamento.

perché è uno dei problemi più studiati per cui esiste una grande varietà di algoritmi di ordinamento

perché molte idee usate negli algoritmi che studieremo possono anche essere utilizzate per risolvere altri problemi

Algoritmi di ordinamento

Studieremo vari algoritmi di ordinamento,

perché con i dati ordinati molti problemi diventano più facili da risolvere, per esempio:

la ricerca di un elemento in una lista

la ricerca di duplicati

la ricerca della coppia di elementi più vicini, cioè a differenza in valore assoluto minima.

solo per citarne alcuni.

Algoritmi di ordinamento

Studieremo vari algoritmi di ordinamento,

perché possiamo usarli come esempi non banali

per le prove di correttezza e

per illustrare le tecniche di valutazione del tempo di calcolo di un algoritmo

Non solo ordinamenti!

Introdurremo gli strumenti matematici che consentono di semplificare l'analisi degli algoritmi, sia nel caso iterativo che ricorsivo.

Studieremo alcune strutture dati fondamentali.

Strutture dati (concrete)

Definizione: una struttura dati è un modo di organizzare i dati in memoria.

Nessun algoritmo può prescindere dall'organizzazione dei dati in memoria.

Nel caso degli ordinamenti e della ricerca si fa riferimento all'array.

Arrays

A =

4	18	22	15	9	7	2
0	1	2	3	4	5	6

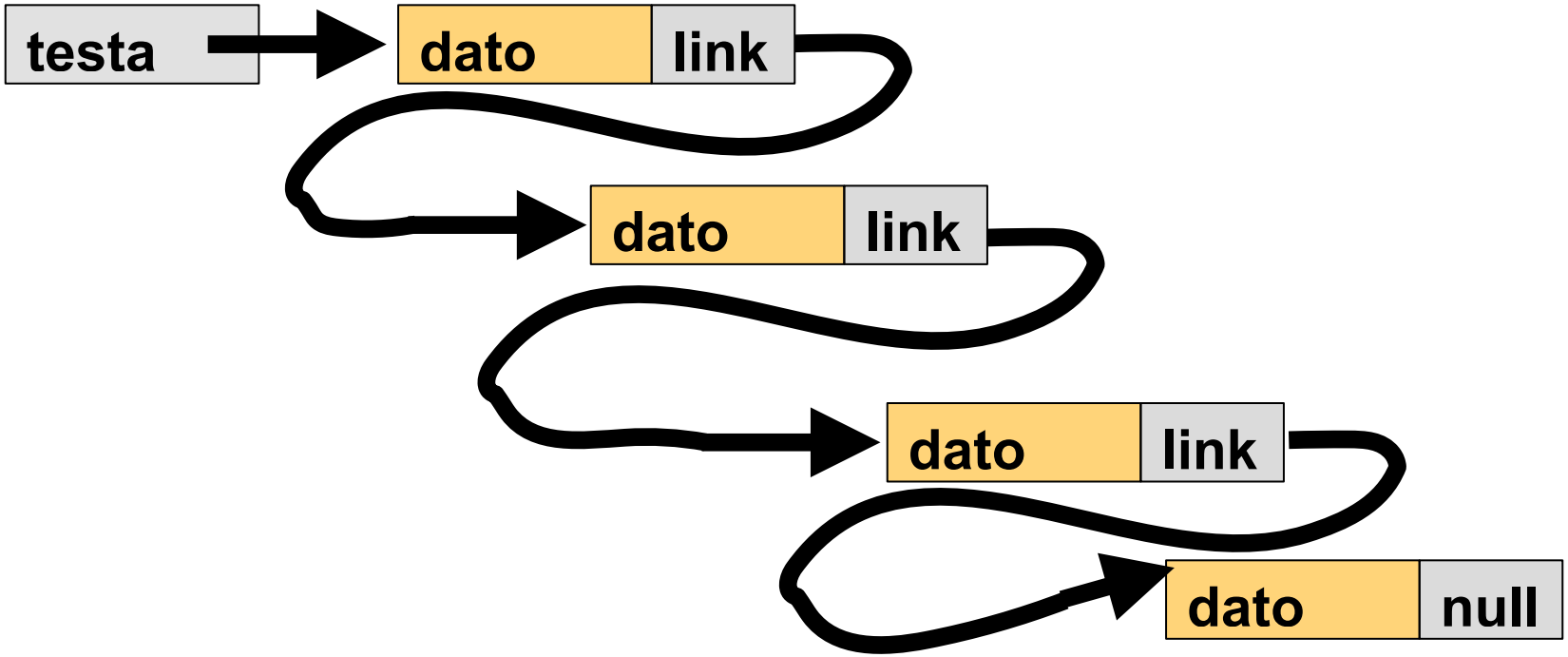
elementi di A

indici degli elementi A

L'elemento A[3] è 15, è il quarto elemento nell'array ed è l'elemento di indice 3.
A differenza della lista di Python, un array può contenere solo elementi dello stesso tipo, è una struttura dati omogenea.
La sua dimensione (lunghezza) è il numero degli elementi. Noi trattiamo array statici, la cui dimensione è quindi impostata alla creazione dell'array stesso.

Liste concatenate

In molti casi un array non basta e servono strutture dati più flessibili, come le liste concatenate:



Strutture dati astratte

In molti casi è utile definire non solo un supporto in memoria per i dati, ma anche fornire delle operazioni per agire su questi dati.

Una struttura dati astratta non è altro che un insieme di elementi con delle operazioni.

Un struttura dati astratta si implementa scegliendo una struttura dati concreta e gli algoritmi per realizzare le operazioni.

Studieremo particolari strutture dati astratte come pile, code e code di priorità.

Pile

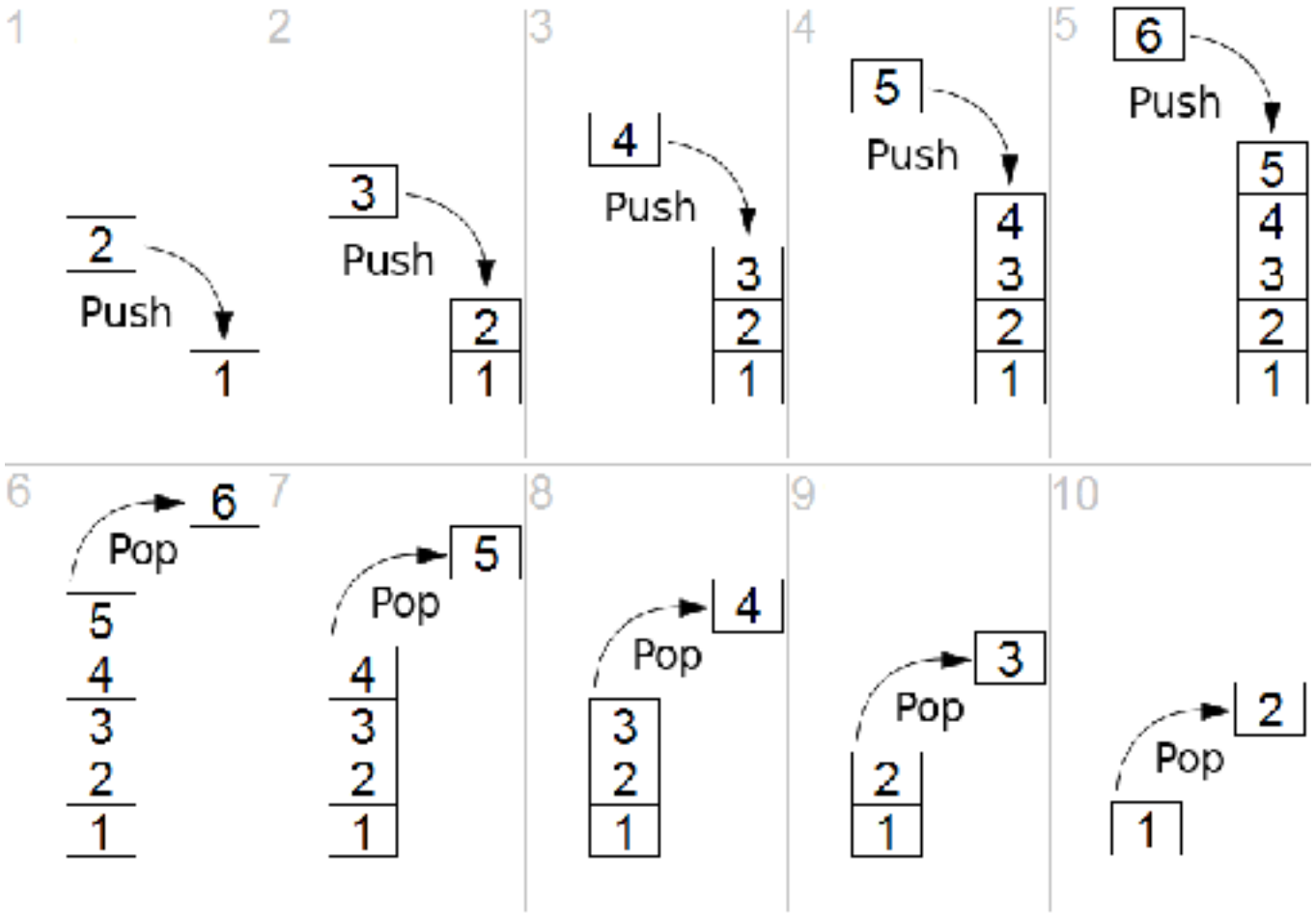
Pila

Una pila è una struttura dati astratta in cui gli elementi sono gestiti seguendo la regola LIFO (Last In First Out).

L'elemento in cima alla pila è l'unico accessibile in lettura o estrazione (pop) e anche l'inserimento avviene esclusivamente in cima (push).

Si implementa memorizzando gli elementi in un array o in una lista concatenata

Pila



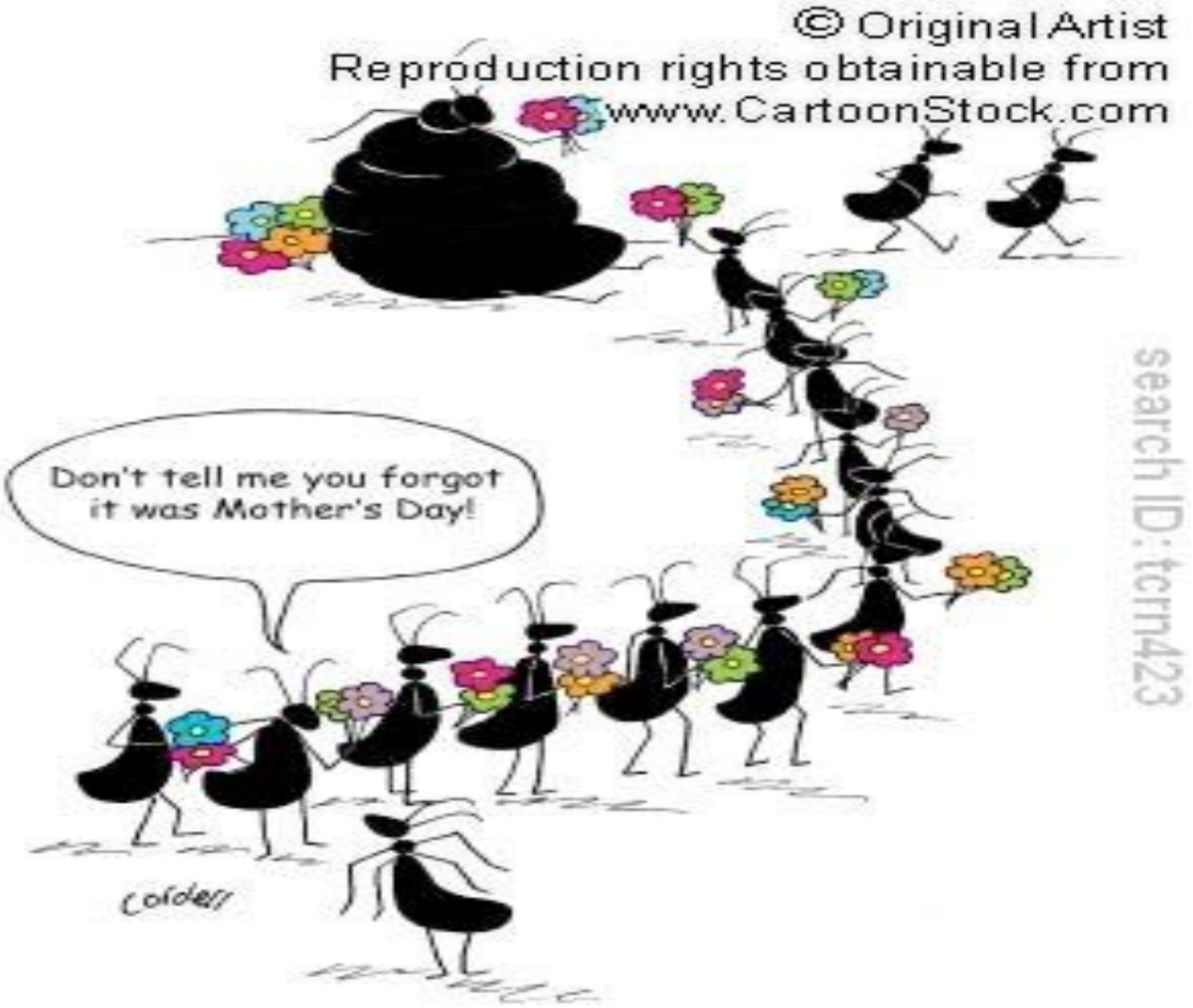
L'immagine da https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Code

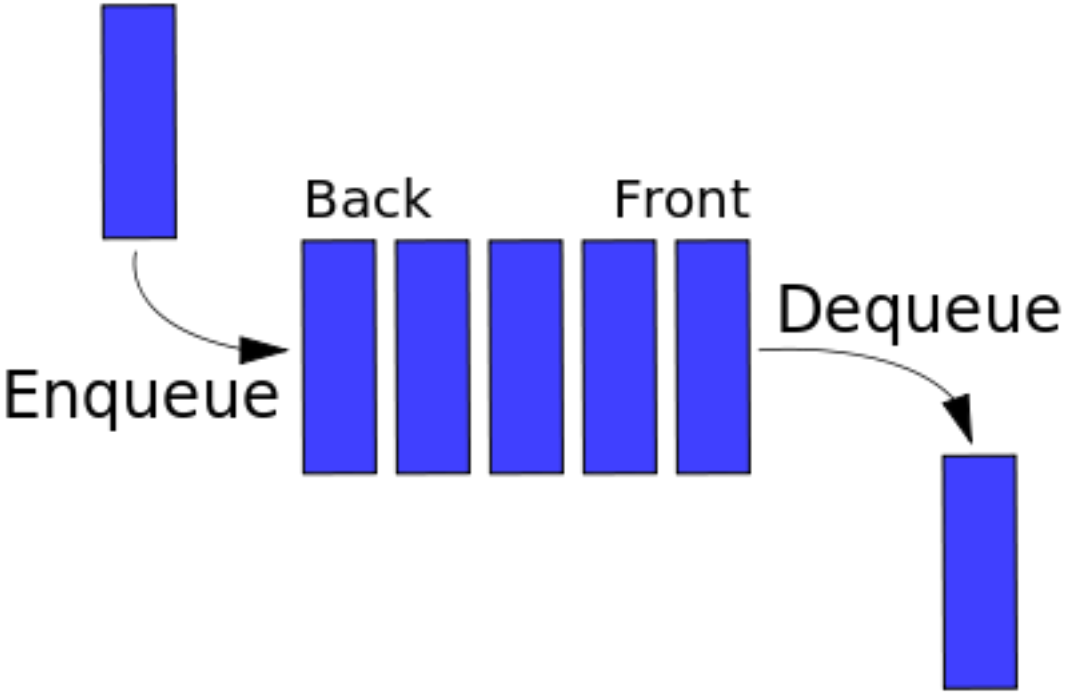
Una coda è una struttura dati astratta in cui gli elementi sono gestiti seguendo la regola FIFO (First In First Out).
Il primo elemento inserito è l'unico accessibile in lettura o estrazione (dequeue) mentre l'inserimento avviene esclusivamente alla fine della coda (enqueue).

Come per la pila si implementa con arrays o liste concatenate

Mettiti in coda …



Code



Code di priorità

Una coda di priorità è una struttura dati astratta in cui gli elementi sono gestiti in dipendenza di una quantità assegnata a ogni elemento detta priorità. L'estrazione di un elemento è fatta sulla base della sua priorità.

L'immagine è presa da: http://www.dalecarnegiewayindy.com/2011/09/23/leadership-training-and-business-success-go-hand-in-hand/

Dizionari

Un dizionario (o tabella dei simboli) è una struttura dati astratta:
un insieme con le operazioni di

- **ricerca**
- **inserimento**
- **cancellazione**

Come le pile, le code, le code di priorità è una struttura dati astratta fondamentale.

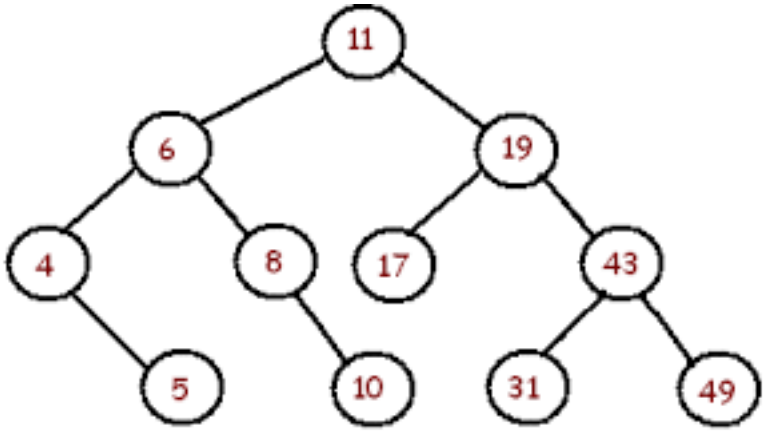
Strutture dati lineari e non

Array e liste concatenate sono strutture dati lineari.

Gli alberi ci offrono la possibilità di organizzare i dati gerarchicamente

Alberi binari

Un albero binario è un particolare tipo di albero radicato.
Ogni nodo ha al più due figli, il sinistro e il destro



Alberi

Gli alberi sono molto utili in computer science:

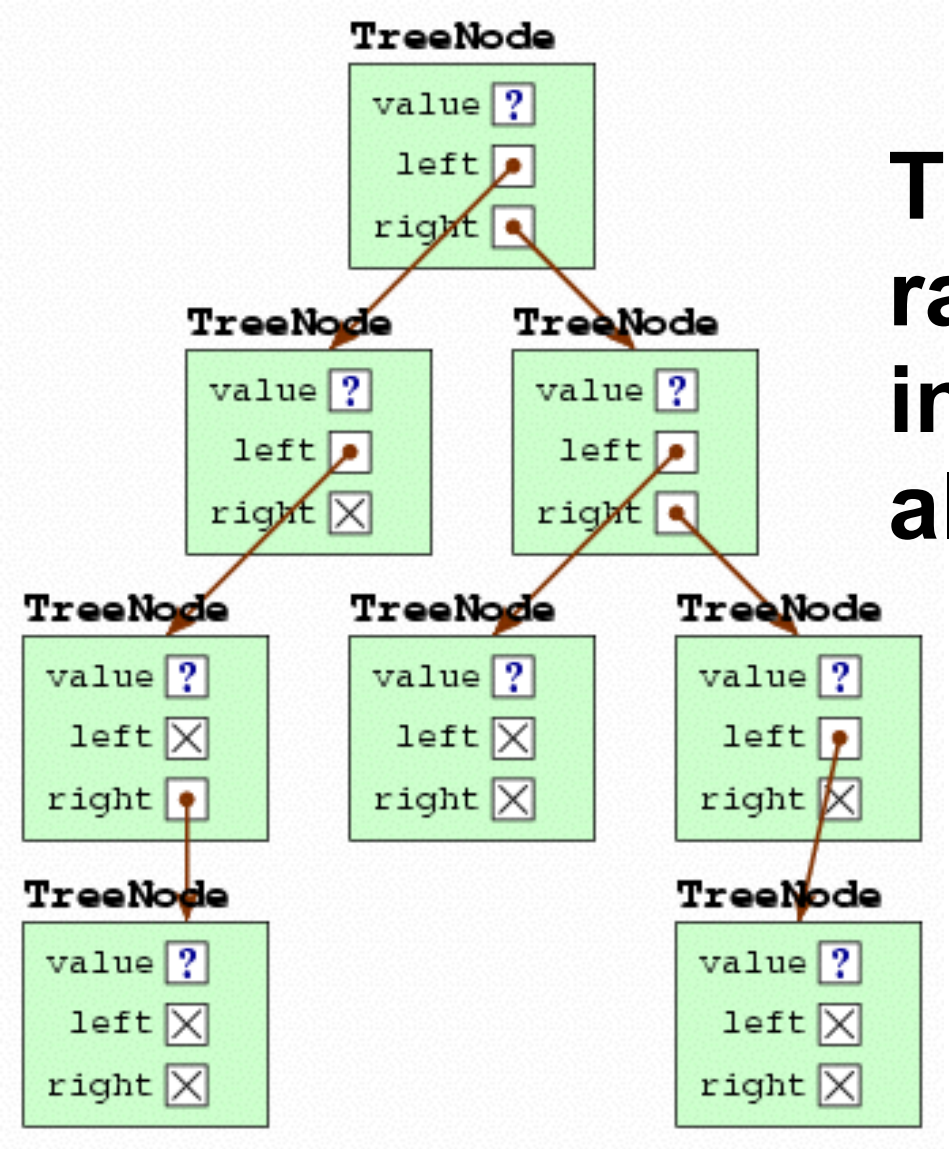
- **come strutture dati in molteplici applicazioni**
- **nell'analisi degli algoritmi**

Alberi in questo corso

I dizionari si possono implementare memorizzando i dati nei nodi di un particolare tipo di albero binario.

Una coda di priorità è implementata su un array, ma "visto" come un particolare tipo di albero binario, in modo da facilitare l'analisi delle operazioni.

Rappresentazione in memoria



Tipica rappresentazione in memoria di un albero binario

L’arte del problem solving

Lo studio degli algoritmi e le strutture dati di questo corso è anche un allenamento mentale al problem solving: l’arte di costruire nuove soluzioni algoritmiche per i problemi

Problem solving

Definizione del problema

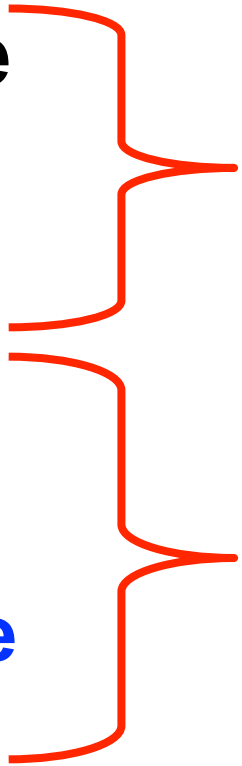
Progetto dell'algoritmo che lo risolve

Analisi dell'algoritmo (correttezza, complessità di tempo e di spazio)

Implementazione in un linguaggio di programmazione

Testing

[manutenzione]



Il libro

T.H. Cormen, C.E. Leiserson, R.L. Rivest e C. Stein , Introduzione agli algoritmi e strutture dati 2/ed, Mc Graw Hill, 2005,

T.H. Cormen, C.E. Leiserson, R.L. Rivest e C. Stein , Introduction to algorithms 3/ed, MIT Press, 2009.

Altre fonti in rete

Giancarlo Bongiovanni e Tiziana Calamoneri, dispense per il corso di Informatica generale:

http://twiki.dsi.uniroma1.it/twiki/view/Infogen/DispenseELibriDiTesto

per le implementazioni in python, e non solo, il sito del testo "Problem solving with algorithms and data structures using Python" di B.N. Miller e D.L. Ranum:

http://interactivepython.org/runestone/static/pythonds/ index.html

per le implementazioni in java, non solo, il sito del testo "Algoritmi in Java" di R. Sedgewick:

https://algs4.cs.princeton.edu/home/

Gli esami

5 appelli: giugno, luglio, settembre, gennaio e febbraio.
Sempre nella prima e quarta settimana dopo la fine delle lezioni

Modalità: l'esame consiste in una prova scritta ed un'eventuale prova orale.

La prova scritta prevede tre esercizi e il tempo a disposizione per risolverli sarà di due ore e mezza.

Gli esami - 2

Prova intermedia: E' prevista una prova a metà corso sulla prima parte del programma e una a fine corso sulla seconda parte.

Ognuna delle due prove conterrà 3 esercizi e il tempo a disposizione per risolverli sarà di 2 ore.

Negli appelli di giugno e luglio chi abbia superato una delle due prove di esonero potrà recuperare quella mancante sostenendo solo la prova relativa alla parte insufficiente.

Esercizi ed esoneri

La prima prova di esonero si terrà il 19 aprile 2018.

La prima esercitazione in vista dell'esonero in aula si terrà il 22 marzo.

La seconda prova di esonero si terrà a fine corso.

La seconda esercitazione in vista dell'esonero in aula si terrà il 10 maggio.