

# ESERCIZI DA SVOLGERE

Laboratorio di Algoritmi e Strutture Dati

# ISTRUZIONI GENERALI

- Il progetto di laboratorio va inizializzato "clonando" il repository su git
- l'uso corretto di git (commit adeguate da parte di tutti i componenti del gruppo) è parte della valutazione
- SU GIT DOVRÀ ESSERE CARICATO SOLAMENTE IL CODICE: NESSUN FILE DATI DOVRÀ ESSERE OGGETTO DI COMMIT!!!!
- Relazione (circa una pagina) sui risultati del primo esercizio

# ISTRUZIONI GENERALI

- Linguaggio da usare: a scelta tra C e Java, apprezzato se usati entrambi
- Uso di librerie esterne e/o native del linguaggio scelto:
  - vietato l'uso di strutture dati di base (es. code, liste, stack, ...), ad eccezione di ArrayList
  - È lecito (ma non obbligatorio) avvalersi di strutture dati più complesse quando la loro realizzazione non è richiesta da uno degli esercizi proposti
    - Es. HashTable OK, libreria per i grafi NO (vedi esercizio 4)



# UNIT TEST

- Implementare gli unit-test di **tutti** gli algoritmi implementati

# DATASET

- Per alcuni esercizi sono forniti files con dataset per effettuare opportune prove
  - /usr/NFS/Linux/labalgoritmi/datasets/
  - in laboratorio von Neumann, selezionare il disco Y
- **NOTA:** questi files non devono essere oggetto di commit su Git!

# ESERCIZIO I

- Implementare una libreria che offra i seguenti algoritmi di ordinamento:
  - insertion sort
  - merge sort



# ESERCIZIO I

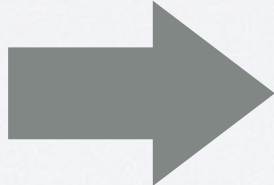
- Ogni algoritmo va implementato in modo tale da poter essere utilizzato su un generico tipo T
- L'implementazione degli algoritmi deve permettere di specificare il criterio secondo cui ordinare i dati
- Suggerimento: Usare l'interfaccia `java.util.Comparator` (o, nel caso di una implementazione C, un puntatore a funzione)

# ESERCIZIO I: PRIMO USO

- File integers.txt con 20 milioni di interi da ordinare
- Gli interi sono scritti di seguito, ciascuno su una riga
- Implementare un'applicazione che, usando ciascuno degli algoritmi di ordinamento offerti dalla libreria, ordina in modo crescente gli interi contenuti nel file



# ESERCIZIO I: PRIMO USO

- Si misurino i tempi di risposta e si crei una breve relazione (circa una pagina) in cui si riportano i risultati ottenuti insieme a un loro commento
- Tempo > 10 minuti  interrompere l'esecuzione e riportare un fallimento dell'operazione
- Commentare nella relazione i risultati

# ESERCIZIO I: SECONDO USO

- Implementare una funzione che:
  - accetta in input un intero  $N$  e un qualunque array  $A$  di interi
  - verifica se  $A$  contiene due interi la cui somma è esattamente  $N$
- Funzione **DEVE** avere complessità  $\Theta(K \log K)$  ( $K$ = numero di elementi di  $A$ )

# ESERCIZIO 1: SECONDO USO

- File sums.txt con 100 numeri interi
  - Gli interi sono scritti di seguito, ciascuno su una riga
- Implementare un'applicazione che carica in un array A gli interi contenuti nel file integers.txt e, per ciascun intero N contenuto nel file sums.txt, verifica se esso è la somma di due elementi contenuti in A
- Si aggiunga un commento sulle prestazioni di questo algoritmo alla relazione scritta per la prima parte dell'esercizio



# ESERCIZIO 2

- Si consideri il problema di determinare la distanza di edit tra due stringhe (Edit distance)
  - date due stringhe  $s1$  e  $s2$ , non necessariamente della stessa lunghezza, determinare il minimo numero di operazioni necessarie per trasformare la stringa  $s2$  in  $s1$
  - Operazioni disponibili:
    - cancellazione di un carattere
    - inserimento di un carattere

# ESERCIZIO 2

- Esempi:

- "casa" e "cassa" edit distance = 1 (1 cancellazione)
- "casa" e "cara" edit distance = 2 (1 cancellazione + 1 inserimento)
- "tassa" e "passato" edit distance = 4 (3 cancellazioni + 1 inserimento)
- "pioppo" e "pioppo" edit distance = 0

# ESERCIZIO 2

- Si implementi una versione ricorsiva della funzione **edit\_distance**
- Sia  $|s|$  la lunghezza di una stringa
- Sia  $\text{rest}(s)$  la sottostringa di  $s$  ottenuta ignorando il primo carattere di  $s$ 
  - se  $|s1| = 0$ , allora  $\text{edit\_distance}(s1, s2) = |s2|$
  - se  $|s2| = 0$ , allora  $\text{edit\_distance}(s1, s2) = |s1|$



# ESERCIZIO 2

- Altrimenti siano:

- $d_{\text{no-op}} = \begin{cases} \text{edit\_distance}(\text{rest}(s1), \text{rest}(s2)) & \text{se } s1[0] = s2[0] \\ \infty & \text{altrimenti} \end{cases}$

- $d_{\text{canc}} = 1 + \text{edit\_distance}(s1, \text{rest}(s2))$

- $d_{\text{ins}} = 1 + \text{edit\_distance}(\text{rest}(s1), s2)$

- Allora:

$$\text{edit\_distance}(s1, s2) = \min\{d_{\text{noop}}, d_{\text{canc}}, d_{\text{ins}}\}$$

# ESERCIZIO 2

- Si implementi una versione **edit\_distance\_dyn** della funzione, adottando una strategia di programmazione dinamica
- **Nota:** Le definizioni alle slides precedenti non corrispondono al modo usuale di definire la distanza di edit, né si prestano ad una implementazione iterativa particolarmente efficiente. Sono del tutto sufficienti però per risolvere l'esercizio e sono quelle su cui dovrete basare la vostra risposta.

# ESERCIZIO 2 - USO DELLE FUNZIONI

- File dictionary.txt
  - elenco delle parole italiane (molte)
  - Parole scritte di seguito, ciascuna su una riga
- File correctme.txt
  - citazione di John Lennon
  - presenti alcuni errori di battitura



# ESERCIZIO 2 - USO DELLE FUNZIONI

- Si implementi un'applicazione che usa la funzione `edit_distance_dyn` per determinare, per ogni parola `w` in correctme.txt, la lista di parole in dictionary.txt con edit distance minima da `w`
- Si sperimenti il funzionamento dell'applicazione e si riporti in una breve relazione (circa una pagina) i risultati degli esperimenti

# ESERCIZIO 3

- Si implementi la struttura dati Coda con priorità
- La struttura dati
  - deve gestire tipi generici
  - consentire un numero qualunque e non noto a priori di elementi

# ESERCIZIO 4

- Si implementi una libreria che realizza la struttura dati Grafo in modo che sia ottimale per dati sparsi
- La struttura deve consentire di rappresentare sia grafi diretti che grafi non diretti
  - suggerimento: un grafo non diretto può essere rappresentato usando un'implementazione per grafi diretti in modo che:
    - per ogni arco  $(a,b)$ , etichettato  $w$ , presente nel grafo, è presente nel grafo anche l'arco  $(b,a)$ , etichettato  $w$
  - il grafo dovrà mantenere l'informazione che specifica se è diretto o no



# ESERCIZIO 4

- Implementare:
  - le funzioni essenziali per la struttura dati Grafo
  - una funzione che restituisce il peso del grafo
    - se il grafo non è pesato, la funzione può terminare con un errore

# ESERCIZIO 4 - USO DELLA LIBRERIA

- Si implementi l'algoritmo di Prim per la determinazione della minima foresta ricoprente del grafo
- utilizzare la struttura dati Coda con priorità dell'esercizio 3
- Se il grafo è costituito da una sola componente connessa, l'algoritmo restituirà un albero, altrimenti restituirà una foresta costituita dai minimi alberi ricoprenti di ciascuna componente connessa

# ESERCIZIO 4 - USO DELLA LIBRERIA E DI PRIM

- Utilizzare l'algoritmo di Prim sul file italian\_dist\_graph.csv
  - distanze in metri tra varie località italiane
  - Formato CSV standard: campi separati da virgole, record separati da fine riga (\n)
- Ogni record contiene i seguenti dati:
  - località 1: (tipo stringa) nome della località "sorgente". La stringa può contenere spazi, non può contenere virgole;
  - località 2: (tipo stringa) nome della località "destinazione". La stringa può contenere spazi, non può contenere virgole;
  - distanza: (tipo float) distanza in metri tra le due località.



# ESERCIZIO 4 - USO DELLA LIBRERIA E DI PRIM

- Interpretare le informazioni del file come archi non diretti
- il file è stato creato a partire da un dataset poco accurato
  - Dati riportati contengono inesattezze e imprecisioni
- Risultato atteso: minima foresta ricoprente con 18.640 nodi, 18.637 archi (non orientati) e di peso complessivo di circa 89.939,913 Km