Laboratorio per il corso di Algoritmi e Strutture Dati: informazioni generali

Il presente documento riporta i testi degli esercizi da svolgere e consegnare al fine di poter sostenere la prova di laboratorio per il corso di Algoritmi e Strutture Dati, preceduti da alcune indicazioni e suggerimenti relativi allo svolgimento degli stessi.

Nota: il presente documento contiene alcune formule descritte usando la sintassi LaTeX. È possibile convertirlo in un pdf di più facile lettura usando l'utilty pandoc. Da riga di comando (Unix):

```
pandoc README.md -o README.pdf
```

Importante:

Gli esercizi sono divisi in una parte di sviluppo di una struttura dati e/o di un algoritmo e una (eventuale) parte in cui si esegue il codice sviluppato su un dataset dato. È importante tenere presente che nello sviluppare la prima parte degli esercizi si deve assumere di stare sviluppando una libreria generica intesa come fondamento di futuri programmi. Non è pertanto lecito fare assunzioni semplificative legate alla particolare applicazione della libreria ai dati forniti. Ad esempio non si può assumere che l'input di un algoritmo di ordinamento sia un vettore di un particolare tipo di elementi o che i valori dei nodi di un grafo siano necessariamente di tipo stringa.

In alcuni esercizi si ribadisce la necessità di implementare una versione generale della libreria. Ciò non vuol dire che dove questo non sia specificato esplicitamente sia lecita una implementazione meno generale.

Durante l'esame sarete chiamati a difendere la generalità dell'implementazione proposta.

Regole per l'esame

Il progetto di laboratorio va consegnato mediante Git (vedi sezione successiva) *entro e non oltre* la data della prova scritta che si intende sostenere. E' vietato sostenere la prova scritta in caso di mancata consegna del progetto di laboratorio. In caso di superamento della prova scritta, la prova orale (discussione del laboratorio) va sostenuta, previa prenotazione mediante apposita procedura che sarà messa a disposizione sulla pagina *i-learn del corso*, nell'*appello orale immediatamente successivo alla prova scritta superata.*

Esempio: - lo studente X sostiene la prova scritta del 14 giugno 2018 - lo studente X deve assicurarsi che il progetto su GitLab sia aggiornato alla versione che vuole presentare al docente di laboratorio; - se lo studente X supera la prova scritta

del 14 giugno 2018 *deve* iscriversi all'appello orale immediatamente successivo del 18 giugno 2018 e prenotarsi su *i-learn* in uno degli slot messi a disposizione dal docente del turno di appartenenza.

Le regole si applicano al *singolo* studente. Ad esempio, si consideri un gruppo di laboratorio costituito dagli studenti X, Y e Z, e si supponga che i soli X e Y sostengano la prova scritta del 14 giugno 2018, X con successo, mentre Y con esito insufficiente. Devono essere rispettate le seguenti condizioni: - il progetto di laboratorio del gruppo deve essere aggiornato alla versione che si intende presentare; - il solo studente X deve sostenere la prova orale del 18 giugno 2018 procedendo come indicato nell'esempio, mentre Y e Z sosterranno la discussione quando avranno superato la prova scritta. Gli studenti Y e Z dovranno, di norma, discutere la stessa versione del progetto di laboratorio che ha discusso lo studente X (i.e., eventuali modifiche al laboratorio succesive alla discussione di X dovranno essere *debitammente documentate (i.e., il log delle modifiche dovrà comparire su GitLab) e motivate).*

Validità del progetto di laboratorio: le specifiche per il progetto di laboratorio descritte in questo documento resteranno valide fino a Gennaio 2019 (i.e, all'ultimo appello dell'ultima sessione relativa al corrente anno accademico) e non oltre!"

Uso di Git

Durante la scrittura del codice è richiesto di usare in modo appropriato il sistema di versioning *Git*. Questa richiesta implica quanto segue:

- il progetto di laboratorio va inizializzato "clonando" il repository del laboratorio come descritto nel file Git.md;
- come è prassi nei moderni ambienti di sviluppo, è richiesto di effettuare commit frequenti. L'ideale è un commit per ogni blocco di lavoro terminato (es. creazione e test di una nuova funzione, soluzione di un baco, creazione di una nuova interfaccia, ...);
- ogni membro del gruppo dovrebbe effettuare il commit delle modifiche che lo hanno visto come principale sviluppatore;
- al termine del lavoro si dovrà consegnare l'intero repository.

Il file `Git.md` contiene un esempio di come usare Git per lo sviluppo degli esercizi proposti per questo laboratorio.

N.B. SU GIT DOVRÀ ESSERE CARICATO SOLAMENTE IL CODICE SORGENTE, IN PARTICOLARE NESSUN FILE DATI DOVRÀ ESSERE OGGETTO DI COMMIT!

Si rammenta che la valutazione del progetto di laboratorio considererà anche l'uso adeguato di git da parte di ciascun membro del gruppo.

Unit testing

Come indicato esplicitamente nei testi degli esercizi, il progetto di laboratorio comprende anche la definizione di opportune suite di unit tests. Si rammenta, però, che il focus del laboratorio è l'implementazione di strutture dati e algoritmi. Relativamente agli unit-test sarà quindi sufficiente che gli studenti dimostrino di averne colto il senso e di saper realizzare una suite di test sufficiente a coprire i casi più comuni (compresi, in particolare, i casi limite).

Linguaggio in cui sviluppare il laboratorio

È lasciata libertà allo studente di implementare il codice usando Java o C. Come potrete verificare gli esercizi chiedono di realizzare strutture generiche. Seguono alcuni suggerimenti sul modo di realizzarle nei due linguaggi accettati.

Nota importante: Con "strutture dati generiche" si fa riferimento al fatto che le strutture dati realizzate devono poter essere utilizzate con tipi di dato non noti a tempo di compilazione. Sebbene in Java la soluzione più in linea con il moderno utilizzo del linguaggio richiederebbe la creazione di classi parametriche, tutte le scelte implementative (compresa la decisione di usare o meno classi parametriche) sono lasciate agli studenti.

Suggerimenti (C): Nel caso del C, è necessario capire come meglio approssimare l'idea di strutture generiche utilizzando quanto permesso dal linguaggio. Un approccio comune è far sì che le funzioni che manipolano le strutture dati prendano in input puntatori a void e utilizzino qualche funzione fornita dall'utente per accedere alle componenti necessarie. *Nota*: chi è in grado di realizzare tipi di dato astratto tramite tipi opachi è incoraggiato a procedere in questa direzione.

Suggerimenti (Java): Nel caso si scelga di utilizzare Java, è possibile (e consigliato) usare gli ArrayList invece degli array nativi al fine di semplificare l'implementazione delle strutture generiche.

Uso di librerie esterne e/o native del linguaggio scelto

A parte gli ArrayList è vietato usare altre strutture dati *di base* offerte dal linguaggio in uso (es. code, liste, stack, ...). È lecito (ma non obbligatorio) avvalersi di strutture dati più complesse quando la loro realizzazione non è richiesta da uno degli esercizi proposti. Ad esempio è lecito utilizzare una libreria che implementa un dizionario (e.g., HashTable), ma non una che implementa i grafi (poiché questi ultimi sono oggetto di un esercizio).

Qualità dell'implementazione

È parte del mandato degli esercizi la realizzazione di codice di buona qualità. Per "buona qualità" intendiamo codice ben modularizzato, ben commentato e ben testato.

Alcuni suggerimenti:

- verificare che il codice sia suddiviso correttamente in package o moduli;
- aggiungere un commento, prima di una definizione, che spiega il funzionamento dell'oggetto definito. Evitare quando possibile di commentare direttamente il codice in sé (se il codice è ben scritto, i commenti in genere non servono);
- la lunghezza di un metodo/funzione è in genere un campanello di allarme: se essa cresce troppo, probabilmente è necessario rifattorizzare il codice spezzando la funzione in più parti. In linea di massima si può consigliare di intervenire quando la funzione cresce sopra le 30 righe (considerando anche commenti e spazi bianchi);
- sono accettabili commenti in italiano, sebbene siano preferibili in inglese;
- tutti i nomi (e.g., nomi di variabili, di metodi, di classi, etc.) devono essere significativi e in inglese;
- il codice deve essere correttamente indentato; impostare l'indentazione a 2 caratteri (un'indentazione di 4 caratteri è ammessa ma scoraggiata) e impostare l'editor in modo che inserisca "soft tabs" (i.e., deve inserire il numero corretto di spazi invece che un carattere di tabulazione).
- per dare i nomi agli identificatori, seguire le convenzioni in uso per il linguaggio scelto:
- Java: i nomi dei package sono tutti in minuscolo senza separazione fra le parole; i nomi dei tipi (classi, interfacce, ecc.) iniziano con una lettera maiuscola e proseguono in camel case (es. TheClass), i nomi dei metodi e delle variabili iniziano con una lettera minuscola e proseguono in camel case (es. theMethod), i nomi delle costanti sono tutti in maiuscolo e in formato snake case (es. THE_CONSTANT);
- C: macro e costanti sono tutti in maiuscolo e in formato snake case (es. THE_MACRO, THE_CONSTANT); i nomi di tipo (e.g. struct, typedefs, enums, ...) iniziano con una lettera maiuscola e proseguono in camel case (e.g., TheType, TheStruct); i nomi di funzione iniziano con una lettera minuscola e proseguono in snake case (e.g., the_function());
- i file vanno salvati in formato UTF8.

Esercizio 1

Implementare una libreria che offre i seguenti algoritmi di ordinamento:

- Insertion Sort

- Merge Sort

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).

Unit testing

Implementare gli unit-test degli algoritmi secondo le indicazioni suggerite nel documento Unit Testing.

Uso della libreria di ordinamento implementata

Ordinamento di un array di interi

Il file `integers.txt` che potete trovare seguendo il path

```
/usr/NFS/Linux/labalgoritmi/datasets/
```

(in laboratorio von Neumann, selezionare il disco Y) contiene 20 milioni di interi da ordinare. Gli interi sono scritti di seguito, ciascuno su una riga. *Gli interi variano su un range molto ampio* (interi a 32 bit potrebbero non essere in grado di rappresentare adeguatamente i valori contenuti nel file).

Implementare un'applicazione che, usando ciascuno degli algoritmi di ordinamento offerti dalla libreria, ordina in modo crescente gli interi contenuti nel file `integers.txt`.

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. Nel caso l'ordinamento si protragga per più di 10 minuti potete interrompere l'esecuzione e riportare un fallimento dell'operazione. I risultati sono quelli che vi sareste aspettati? Se sì, perché? Se no, fate delle ipotesi circa il motivo per cui gli algoritmi non funzionano come vi aspettate, verificatele e riportate quanto scoperto nella relazione.

IL FILE `integers.txt` NON DEVE ESSERE OGGETTO DI COMMIT SU GIT!

Implementazione di un test su array di interi

Implementare una funzione che accetta in input un intero N e un qualunque array A di interi e che verifica se A contiene due interi la cui somma è esattamente N. La funzione DEVE avere complessità $\Theta(KlogK)$ sul numero K di elementi dell'array A.

Il file `sums.txt` che potete trovare seguendo il path

`/usr/NFS/Linux/labalgoritmi/datasets/`

(in laboratorio von Neumann, selezionare il disco Y) contiene 100 interi. Gli interi sono scritti di seguito, ciascuno su una riga.

Come esempio di uso della funzione, 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 alla relazione scritta per l'esercizio precedente un paragrafo in cui si riportano i risultati relativi a questo esercizio.

IL FILE `sums.txt` NON DEVE ESSERE OGGETTO DI COMMIT SU GIT!

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. Si assuma che le operazioni disponibili siano soltanto due: cancellazione e inserimento di un carattere. Esempi:

- "casa" e "cassa" hanno edit distance pari a 1 (1 cancellazione);
- "casa" e "cara" hanno edit distance pari a 2 (1 cancellazione + 1 inserimento);
- "tassa" e "passato" hanno edit distance pari a 4 (3 cancellazioni + 1 inserimento);
- "pioppo" e "pioppo" hanno edit distance pari a 0.

Si implementi una versione ricorsiva della funzione `edit_distance` basata sulle seguenti osservazioni (indichiamo con $|s|$ la lunghezza di s e con rest(s) la sottostringa di s ottenuta ignorando il primo carattere di s):

- se $|s1| = 0$, allora edit_distance$(s1, s2) = |s2|$;

- se $|s2| = 0$, allora edit_distance$(s1, s2) = |s1|$;
- altrimenti, siano:
- $d_{\text{no-op}} = \begin{cases} \text{edit\_distance}(\text{rest}(s1), \text{rest}(s2)) & \text{se } s1[0] = s2[0] \\ \infty & \text{altrimenti} \end{cases}$
- $d_{\text{canc}} = 1 + \text{edit\_distance}(s1, \text{rest}(s2))$
- $d_{\text{ins}} = 1 + \text{edit\_distance}(\text{rest}(s1), s2)$

Si ha: edit_distance$(s1, s2) = \min\{d_{\text{no-op}}, d_{\text{canc}}, d_{\text{ins}}\}$

Si implementi una seconda versione `edit_distance_dyn` della funzione, adottando una strategia di programmazione dinamica.

Nota: Le definizioni sopra riportate non corrispondono al modo usuale di definire la distanza di edit, né si prestano a una implementazione iterativa particolarmente efficiente. Sono del tutto sufficienti però per risolvere l'esercizio e sono quelle su cui dovrete basare la vostra risposta.

Unit Testing

Implementare gli unit-test degli algoritmi secondo le indicazioni suggerite nel documento Unit Testing.

Uso delle funzioni implementate

Il file `dictionary.txt` che potete trovare seguendo il path

```
/usr/NFS/Linux/labalgoritmi/datasets/
```

(in laboratorio von Neumann, selezionare il disco Y) contiene l'elenco (di una parte significativa) delle parole italiane. Le parole sono scritte di seguito, ciascuna su una riga.

Il file `correctme.txt` contiene una citazione di John Lennon. Il file contiene alcuni errori di battitura.

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.

I FILE `dictionary.txt` E `correctme.txt` NON DEVONO ESSERE OGGETTO DI COMMIT SU GIT!

Esercizio 3

Si implementi la struttura dati Coda con priorità.

La struttura dati deve gestire tipi generici e consentire un numero qualunque e non noto a priori di elementi.

Unit Testing

Implementare gli unit-test degli algoritmi secondo le indicazioni suggerite nel documento Unit Testing.

Esercizio 4

Si implementi una libreria che realizza la struttura dati Grafo in modo che sia ottimale per dati sparsi (IMPORTANTE: le scelte implementative che farete dovranno essere giustificate in relazione alle nozioni presentate durante le lezioni in aula). 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 nel seguente modo: per ogni arco `(a,b)`, etichettato `w`, presente nel grafo, è presente nel grafo anche l'arco `(b,a)`, etichettato `w`. Ovviamente, il grafo dovrà mantenere l'informazione che specifica se esso è un grafo diretto o non diretto.).

Oltre alle funzioni essenziali per la struttura dati Grafo, si implementi nella libreria anche una funzione che restituisce il peso del grafo (se il grafo non è pesato, la funzione può terminare con un errore).

Unit Testing

Implementare gli unit-test degli algoritmi secondo le indicazioni suggerite nel documento Unit Testing.

Uso della libreria che implementa la struttura dati Grafo

Si implementi l'algoritmo di Prim per la determinazione della minima foresta ricoprente di un grafo. L'implementazione dell'algoritmo di Prim dovrà utilizzare la struttura dati Coda con priorità nell'esercizio precedente.

N.B. Nel caso in cui il grafo sia costituito da una sola componente connessa, l'algoritmo restituirà un albero; nel caso in cui, invece, vi siano più componenti connesse, l'algoritmo restituirà una foresta costituita dai minimi alberi ricoprenti di ciascuna componente connessa.

Uso delle librerie che implementano la struttura dati Grafo e l'algoritmo di Prim

La struttura dati e l'algoritmo di Prim dovranno essere utilizzati con i dati contenuti nel file `italian_dist_graph.csv`.

Il file `italian_dist_graph.csv` che potete recuperare seguendo il path

`/usr/NFS/Linux/labalgoritmi/datasets/`

(in laboratorio von Neumann, selezionare il disco Y) contiene le distanze in metri tra varie località italiane e una frazione delle località a loro più vicine. Il formato è un CSV standard: i campi sono separati da virgole; i record sono separati dal carattere di 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.

Note:

- potete intrepretare le informazioni presenti nelle righe del file come archi **non diretti** (i.e., probabilmente vorrete inserire nel vostro grafo sia l'arco di andata che quello di ritorno a fronte di ogni riga letta).
- il file è stato creato a partire da un dataset poco accurato. I dati riportati contengono inesattezze e imprecisioni.

IL FILE `italian_dist_graph.csv` NON DEVE ESSERE OGGETTO DI COMMIT SU GIT!

Controlli

Un'implementazione corretta dell'algoritmo di Prim, eseguita sui dati contenuti nel file `italian_dist_graph.csv`, dovrebbe determinare una minima foresta ricoprente con 18.640 nodi, 18.637 archi (non orientati) e di peso complessivo di circa 89.939,913 Km.