Laboratorio per il corso di Algoritmi e Strutture Dati: regole d'esame, indicazioni generali e suggerimenti, consegne per gli esercizi

Regole d'esame

Il progetto di laboratorio può essere svolto individualmente o in gruppo di al più 3 persone. **I membri di uno stesso gruppo devono appartenere tutti allo stesso turno di laboratorio**.

ll progetto di laboratorio va consegnato mediante Git (vedi sotto) 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, **nella medesima sessione della prova scritta superata** (si ricorda che le sessioni sono giugno-luglio 2019, settembre 2019, dicembre 2019 e gennaio-febbraio 2020).

Si noti che, per le sessioni di settembre 2019 e dicembre 2019 esiste una sola possibilità per la discussione del laboratorio. Ad esempio, se lo studente X supera la prova scritta a dicembre 2019, deve necessariamente sostenere la discussione di laboratorio con la prova orale di dicembre 2019 (non sarà possibile discutere a gennaio-febbraio 2020).

Esempio:

- lo studente X sostiene la prova scritta nel primo appello di giugno
- lo studente X deve assicurarsi che il progetto su GitLab, alla data della prova scritta che intende sostenere (in questo esempio, quella del primo appello di giugno), sia aggiornato alla versione che vuole presentare al docente di laboratorio;
- se lo studente X supera la prova scritta nel primo appello di giugno, deve (pena la perdita del voto ottenuto nella prova scritta) iscriversi a uno degli appelli orali di giugno o luglio, prenotarsi su i-learn in uno degli slot messi a disposizione dal docente del turno di appartenenza e sostenere l'orale nello slot temporale prenotato.

Le regole riportate sopra si applicano al singolo studente. Per poter accedere alla discussione di laboratori è in ogni caso necessaria l'iscrizione alla prova orale corrispondente su myunito.

Studenti diversi, appartenenti allo stesso gruppo, possono sostenere la prova **scritta** nello stesso appello o in appelli diversi. Se studenti diversi, appartenenti allo stesso gruppo, superano la prova scritta nello stesso appello, devono sostenere l' **orale** nello stesso appello orale. Se studenti diversi, appartenenti allo stesso

gruppo, superano la prova scritta in appelli diversi, possono sostenere l'orale in appelli diversi.

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 nel primo appello di giugno, X con successo, mentre Y con esito insufficiente. Devono essere rispettate le seguenti condizioni:

- alla data della prova scritta del primo appello di giugno, il progetto di laboratorio del gruppo deve essere aggiornato alla versione che si intende presentare;
- il solo studente X deve sostenere la prova orale nella sessione giugno-luglio, procedendo come indicato nell'esempio riportato sopra, mentre Y e Z sosterranno la discussione quando avranno superato la prova scritta.
- Supponiamo che Y e Z superino la prova scritta nell'appello di gennaio: essi dovranno sostenere la prova orale in uno stesso appello della stessa sessione di gennaio-febbraio
- 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 successive alla discussione di X dovranno essere debitamente 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 all'ultimo appello della sessione gennaio-febbraio relativa al corrente anno accademico **(vale a dire, quella di gennaio-febbraio 2020)** e non oltre!. Gli appelli delle sessioni successive a questa dovranno essere sostenuti sulla base delle specifiche che verranno descritte nella prossima edizione del laboratorio di algoritmi.

Come unica eccezione si ammetterà, per il solo primo appello della sessione giugno-luglio dell'anno accademico successivo a quello corrente (vale a dire per il primo appello di giugno 2020), che venga discusso il laboratorio presentato in questo documento **a patto che i commit su gitlab dimostrino che il lavoro è stato completato entro la sessione di gennaio-febbraio relativa all'anno accademico corrente**.

Indicazioni generali e suggerimenti

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.

Linguaggio in cui sviluppare il laboratorio

Gli esercizi vanno implementati utilizzando il linguaggio C o Java come precisato di seguito:

- Esercizio 1: C
- Esercizio 2: C o Java a discrezione dello studente
- Esercizio 3: Java
- Esercizio 4: Java

Come detto, 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.

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): 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. Inoltre, 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

Nello sviluppo in Java l'uso di ArrayList, ove non escluso esplicitamente dalla consegna dell'esercizio, è da ritenersi possibile.

È, invece, sempre proibito (sia nello sviluppo in Java che in quello in C) l'uso di strutture dati native del linguaggio scelto o offerte da librerie esterne, quando la loro realizzazione è richiesta da uno degli esercizi proposti.

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 (es., nomi di variabili, di metodi, di classi, ecc.) 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" (cioè, 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 (es. thepackage); 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 UTF-8.

Consegne per gli esercizi

Nota : la presente sezione contiene alcune formule descritte usando la sintassi LaTeX. È possibile convertire l'intero documento in formato pdf - di più facile lettura - usando l'utility pandoc. Da riga di comando (Unix):

pandoc README.md -o README.pdf

Importante: Tutti gli esercizi richiedono almeno di sviluppare una struttura dati e/o un algoritmo. Nello sviluppare questa parte, si deve assumere di stare sviluppando una libreria generica intesa come fondamento di futuri programmi. Non è pertanto lecito fare assunzioni semplificative legate ai particolari usi che di tale libreria generica gli esercizi potrebbero richiedere di implementare; in generale, l'implementazione della libreria generica non deve essere influenzata in alcun modo dagli usi di essa eventualmente richiesti negli esercizi (ad esempio, se un esercizio dovesse richiedere l'implementazione della struttura dati grafo e quello stesso o un altro esercizio dovesse richiedere l'implementazione, a partire da tale struttura dati, di un algoritmo per il calcolo delle componenti connesse di un grafo, l'implementazione della struttura dati non dovrebbe contenere elementi – variabili, procedure, funzioni, metodi, ecc. – eventualmente utili per il calcolo delle componenti connesse, ma non essenziali alla struttura dati; analogamente, se un esercizio dovesse richiedere di operare su grafi con nodi di tipo stringa, l'implementazione della struttura dati grafo dovrebbe restare generica e non potrebbe quindi assumere per i nodi il solo tipo stringa).

Importante: In sede di discussione d'esame, sarà facoltà del docente chiedere di eseguire gli algoritmi implementati su dati forniti dal docente stesso. Nel caso questi dati siano memorizzati su file, questi saranno dei csv con la medesima struttura dei dataset forniti e descritti nel testo dell'esercizio. I codici sviluppati dovranno consentire un rapido e semplice adattamento agli input forniti: ad esempio, una buona implementazione consentirà di inserire in input il nome del file su cui eseguire il test, mentre una peggiore richiederà di modificare il codice sorgente e una successiva compilazione a fronte della sola modifica del nome del file contenente il dataset.

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.

Inoltre tutti gli esercizi chiedono di implementare un programma che sfrutta la libreria realizzata. Questa parte degli esercizi (e solo questa) può essere pensata come una istanziazione particolare di un problema e può quindi fare leva sulle caratteristiche particolari del problema (es., può assumere che i dati siano di un particolare tipo).

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).

Esercizio 1

Linguaggio richiesto: C

Testo

Si consideri il tipo di dato astratto Lista, definito nei termini delle seguenti operazioni:

- Verifica se la lista è vuota in O(1)
- Inserimento in coda alla lista in O(1)
- Inserimento di un elemento nella posizione i-esima della lista in O(n)
- Cancellazione dell'elemento in coda alla lista in O(1)
- Cancellazione dell'elemento in posizione i-esima nella lista in O(n)
- Recupero dell'elemento in posizione i-esima nella lista (senza cancellare l'elemento dalla lista) in O(n)
- Recupero del numero di elementi della lista in O(1)
- Creazione di un iteratore per la lista in O(1)

La lista può contenere oggetti di tipo qualunque e non noto a priori.

Un iteratore è un tipo di dato astratto che permette di iterare su un container di qualche tipo. Un iteratore deve mettere a disposizione le seguenti operazioni:

- Verifica se l'iteratore è ancora valido in O(1) (un iteratore è inizializzato in modo da fare riferimento alla testa della lista e diventa invalido quando viene spostato oltre la fine della lista).
- Recupera l'elemento corrente in O(1)
- Sposta l'iteratore all'elemento successivo in O(1)

Si realizzino in C due implementazioni alternative per il tipo di dato astratto Lista (e, conseguentemente per l'iteratore su di essa).

In particolare:

- entrambe le implementazioni devono offrire:
  - una funzione per creare una lista vuota;
  - una per distruggerla (con conseguente deallocazione della memoria associata);
  - una funzione per distruggere un iteratore (con conseguente deallocazione della memoria associata);
  - tutte e sole le operazioni specificate sopra; tali operazioni devono essere realizzate tramite funzioni aventi la stessa signature in entrambe le librerie;
- una implementazione deve realizzare le liste con array dinamici (cioè ridimensionabili); l'altra implementazione deve realizzare le liste tramite record collegati.

Unit Testing

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

Uso delle librerie implementate

Implementare un algoritmo *merge* che accetta in input un criterio di ordinamento e due liste ordinate secondo tale criterio di ordinamento e restituisce in output una nuova lista, corrispondente alla fusione delle due liste di input e ordinata secondo lo stesso criterio.

L'algoritmo implementato deve poter essere eseguito *senza modifiche* su ciascuna delle due implementazioni per il tipo di dato astratto Lista prodotte secondo le specifiche riportate sopra.

Unit Testing

Implementare gli unit-test per la funzione che implementa *merge* secondo le indicazioni suggerite nel documento Unit Testing.

Esercizio 2

Linguaggio richiesto: C o Java

Testo

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: cancellazione, inserimento, e rimpiazzamento di un carattere. Esempi:

- "casa" e "cassa" hanno edit distance pari a 1 (1 cancellazione);
- "casa" e "cara" hanno edit distance pari a 1 (1 rimpiazzamento);
- "vinaio" e "vino" hanno edit distance=2 (2 inserimenti);
- "tassa" e "passato" hanno edit distance pari a 3 (2 cancellazioni + 1 rimpiazzamento);
- "pioppo" e "pioppo" hanno edit distance pari a 0.

1. Si implementi una versione ricorsiva della funzione edit_distance basata sulle seguenti osservazioni (indichiamo con $|s|$ la lunghezza di s e con $\text{rest}(s)$ 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 $\text{edit\_distance}(s1, s2) = |s1|$;
- altrimenti, siano:
- $d_{\text{no-op}} = \begin{cases} \text{edit\_distance}(\text{rest}(s1), \text{rest}(s2)) & \text{se } s1[0] = s2[0] \\ \infty & \text{altrimenti} \end{cases}$
- $d_{\text{canc}} = 1 + \text{edit\_distance}(s1, \text{rest}(s2))$
- $d_{\text{ins}} = 1 + \text{edit\_distance}(\text{rest}(s1), s2)$
- $d_{\text{replace}} = 1 + \text{edit\_distance}(\text{rest}(s1), \text{rest}(s2))$

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

1. Si implementi una seconda versione edit_distance_dyn della funzione, adottando una strategia di programmazione dinamica. Tale versione deve essere anch'essa ricorsiva (in particolare, essa può essere facilmente ottenuta a partire dall'implementazione richiesta al punto precedente).

Nota: Le definizioni sopra riportate non corrispondono al modo usuale di definire la distanza di edit. Sono del tutto sufficienti però per risolvere l'esercizio e sono quelle su cui dovrà essere basato il codice prodotto.

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

Linguaggio richiesto: Java

Testo

Si implementi la struttura dati Union-Find Set. La struttura dati deve permettere di inserire oggetti di tipo generico e non deve prevedere alcuna cardinalità massima per l'insieme iniziale di elementi.

Unit Testing

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

Esercizio 4

Linguaggio richiesto: Java

Testo

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 modificata per garantire che, per ogni arco (a,b), etichettato w, presente nel grafo, sia 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.).

L'implementazione deve essere generica sia per quanto riguarda il tipo dei nodi, sia per quanto riguarda le etichette degli archi.

La struttura dati implementata dovrà offrire (almeno) le seguenti operazioni (accanto ad ogni operazione è specificata la complessità richiesta; n può indicare il numero di nodi o il numero di archi, a seconda del contesto):

- Creazione di un grafo vuoto – O(1)
- Aggiunta di un nodo – O(1)
- Aggiunta di un arco – O(1)
- Verifica se il grafo è diretto – O(1)
- Verifica se il grafo contiene un dato nodo – O(1)
- Verifica se il grafo contiene un dato arco – O(1) quando il grafo è veramente sparso
- Cancellazione di un nodo – O(n)
- Cancellazione di un arco – O(1) quando il grafo è veramente sparso
- Determinazione del numero di nodi – O(n)
- Determinazione del numero di archi – O(n)
- Recupero dei nodi del grafo – O(n)
- Recupero degli archi del grafo – O(n)
- Recupero nodi adiacenti di un dato nodo – O(n)
- Recupero etichetta associata a una coppia di nodi – O(1) quando il grafo è veramente sparso.

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 Kruskal per la determinazione della minima foresta ricoprente di un grafo.

L'implementazione dell'algoritmo di Kruskal dovrà utilizzare la struttura dati Union-Find Set implementata 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 Kruskal

La struttura dati Grafo e l'algoritmo di Kruskal 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 interpretare 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 Kruskal, 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.