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

Regole d'esame

Importante: gli studenti che hanno nel piano di studi l'insegnamento di Algoritmi con un numero di CFU differente dal quello della corrente edizione (es. 12 CFU) sono pregati di contattare il docente al più presto, al fine di concordare un programma d'esame commisurato ai CFU.

Il progetto di laboratorio può essere svolto individualmente o in gruppo (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 2021, settembre 2021, dicembre 2021 e gennaio-febbraio 2022).

Si noti che, per la sola sessione di giugno-luglio saranno previsti due appelli e, pertanto, esisteranno due possibilità per la discussione del laboratorio (primo o secondo appello della sessione). Nelle altre sessioni, l'appello è unico. Ad esempio, se la studentessa/lo studente X supera la prova scritta a dicembre 2021, deve necessariamente sostenere la discussione di laboratorio con la prova orale di dicembre 2021 (non sarà possibile discutere a gennaio-febbraio 2022).

Esempio:

- la studentessa/lo studente X sostiene la prova scritta nel primo appello di giugno;
- la studentessa/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 la studentessa/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 alla singola studentessa/al singolo studente. Per poter accedere alla discussione di laboratori è in ogni caso necessaria l'iscrizione alla prova orale corrispondente su myunito.

Studentesse/studenti diversi, appartenenti allo stesso gruppo, possono sostenere la prova **scritta** nello stesso appello o in appelli diversi. Se studentesse/studenti diversi, appartenenti allo stesso gruppo, superano la prova scritta nello stesso appello, **devono** sostenere l' **orale** nello stesso appello orale. Se studentesse/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 dalle studentesse/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 nella 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 2022)** 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.

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.

Nota importante: 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
- Esercizio 3: Java
- Esercizio 4: Java

Come indicato sotto, alcuni esercizi chiedono di implementare codice generico. Seguono alcuni suggerimenti sul modo di realizzare codice con questa caratteristica nei due linguaggi accettati.

Nota : Con "codice generico" si intende codice che deve poter essere eseguito 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 codice generico utilizzando quanto permesso dal linguaggio. Un approccio comune è far sì che le funzioni e le procedure presenti nel codice 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 la realizzazione di codice generico.

Uso di librerie esterne e/o native del linguaggio scelto

È vietato (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.

È, invece, possibile l'uso di strutture dati native del linguaggio o offerte da librerie esterne, se la loro realizzazione non è richiesta da uno degli esercizi proposti.

Es.: nello sviluppo in Java, l'uso di ArrayList è da ritenersi possibile, se nessun esercizio chiede la realizzazione in Java di un array dinamico.

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 interno alle funzioni/metodi implementati (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: Gli esercizi 1 3 e 4 richiedono (fra le altre cose) di sviluppare codice generico. 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; in generale, l'implementazione della libreria generica deve restare separata e 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 deve essere separata dall'algoritmo per il calcolo delle componenti connesse e *non* deve contenere elementi – variabili, procedure, funzioni, metodi, ecc. – eventualmente utili a tale algoritmo, 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).

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.

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

Implementare una libreria che offre un algoritmo di ordinamento `Merge-BinaryInsertion Sort`. Con `BinaryInsertion Sort` ci riferiamo a una versione dell'algoritmo `Insertion Sort` in cui la posizione, all'interno della sezione ordinata del vettore, in cui inserire l'elemento corrente è determinata tramite ricerca binaria. Il `Merge-BinaryInsertion Sort` è un algoritmo ibrido che combina `Merge Sort` e `BinaryInsertion Sort`. L'idea è di approfittare del fatto che il `BinaryInsertion Sort` può essere più veloce del `Merge Sort` quando la sottolista da ordinare è piccola. Ciò suggerisce di considerare una modifica del `Merge Sort` in cui le sottoliste di lunghezza `k` o inferiore sono ordinate usando il `BinaryInsertion Sort` e sono poi combinate usando il meccanismo tradizionale di fusione del `Merge Sort`. Il valore del parametro `k` dovrà essere studiato e discusso nella relazione. Ad esempio, `k=0` implica che `Merge-BinaryInsertion Sort` si comporta esattamente come il `Merge Sort` classico, mentre `k>>0` aumenta l'utilizzo del `BinaryInsertion Sort`.

Il codice che implementa `Merge-BinaryInsertion Sort` deve essere generico. Inoltre, la libreria deve permettere di specificare (cioè deve accettare in input) il criterio secondo cui ordinare i dati.

Unit Testing

Implementare gli unit-test per la libreria secondo le indicazioni suggerite nel documento Unit Testing.

Uso della libreria di ordinamento implementata

Il file `records.csv` che potete trovare (compresso) all'indirizzo

`https://datacloud.di.unito.it/index.php/s/X7qC8JSLNRtLxPC`

contiene 20 milioni di record da ordinare. Ogni record è descritto su una riga e contiene i seguenti campi:

- id: (tipo intero) identificatore univoco del record;

- field1: (tipo stringa) contiene parole estratte dalla divina commedia, potete assumere che i valori non contengano spazi o virgole;
- field2: (tipo intero);
- field3: (tipo floating point);

Il formato è un CSV standard: i campi sono separati da virgole; i record sono separati da `\n`.

Usando l'algoritmo implementato, si ordinino i *record* (non è sufficiente ordinare i singoli campi) contenuti nel file `records.csv` in ordine non decrescente secondo i valori contenuti nei tre campi "field" (cioè, per ogni valore di `k`, è necessario ripetere l'ordinamento tre volte, una volta per ciascun campo).

Si misurino i tempi di risposta variando il valore di `k` e si produca una breve relazione in cui si riportano i risultati ottenuti insieme a un loro commento. Dimostrare nella relazione come il valore di `k` dovrebbe essere scelto nella pratica. 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 l'algoritmo non funziona come vi aspettate, verificatele e riportate quanto scoperto nella relazione.

Si ricorda che che il file `records.csv` **NON DEVE ESSERE OGGETTO DI COMMIT SU GIT!**

Esercizio 2

Linguaggio richiesto: C

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 e inserimento. 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);
- "vinaio" e "vino" hanno edit distance=2 (2 inserimenti);
- "tassa" e "passato" hanno edit distance pari a 4 (3 cancellazioni + 1 inserimento);
- "pioppo" e "pioppo" hanno edit distance pari a 0.

1. Si implementi una versione ricorsiva della funzione edit_distance la cui struttura riproponga quella della seguente definizione (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$(s1, s2) = \min\{d_{\text{no-op}}, d_{\text{canc}}, d_{\text{ins}}\}$

2. Si implementi una seconda versione edit_distance_dyn della funzione, adottando una strategia di programmazione dinamica. Tale versione deve essere anch'essa ricorsiva e la sua struttura deve essere simile a quella della versione richiesta al punto precedente.

Nota: Le definizioni sopra riportate non corrispondono al modo usuale di definire la distanza di edit. Sono però del tutto sufficienti per risolvere l'esercizio e, come detto, 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

All'indirizzo

`https://datacloud.di.unito.it/index.php/s/gfoEndRSfwQKiHS`

potete trovare i file `dictionary.txt` e `correctme.txt` (in una cartella compressa).

Il file `dictionary.txt` 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. La citazione presenta 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.

Si ricorda che 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 (con le euristiche di unione per rango e compressione del cammino). La struttura dati deve permettere di inserire oggetti di tipo generico e non prevedere un insieme iniziale finito di elementi.

Una descrizione della Union-Find Set è riportata sul testo Cormen et al., `Introduzione agli algoritmi e strutture dati`, McGraw-Hill, nel capitolo `Strutture dati per insiemi disgiunti`, paragrafo `Foreste di insiemi disgiunti`.

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) (*)
- Cancellazione di un nodo – O(n)

- Cancellazione di un arco – O(1) (*)
- Determinazione del numero di nodi – O(1)
- 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(1) (*)
- Recupero etichetta associata a una coppia di nodi – O(1) (*)

(*) quando il grafo è veramente sparso, assumendo che l'operazione venga effettuata su un nodo la cui lista di adiacenza ha una lunghezza in O(1).

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 trovare all'indirizzo

`https://datacloud.di.unito.it/index.php/s/PirTJpq4JMnpH3G`

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 (pertanto, si suggerisce di inserire nel 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.

Si ricorda 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.