Strutture Dati Algoritmi e Complessità **Name:** _____
A.A. 2023/2024
Compito A
20/06/2024
Time Limit: 180 minuti (3 ore) [Studenti DSA: no quesiti 2, 4 e 6]

Questo compito contiene 7 pagine (inclusa questa) e 7 quesiti.
Il numero totale di punti è **32**.

Tabella dei punteggi (per i docenti)

Question	Points	Score
1	10	
2	4	
3	4	
4	4	
5	3	
6	4	
7	3	
Total:	32	

Vi sono 7 quesiti diversi, i cui punteggi ne identificano l'importanza relativa.

Si prega di scrivere in modo chiaro. Completezza (non lungaggine) e rigore nell'esposizione delle vostre argomentazioni sono fattori decisivi ai fini della valutazione. Non occorrse scrivere tanto, ma scrivere bene il necessario.

Istruzioni

Lanciare la macchina virtuale Oracle VirtualBox e lavorare all'interno della cartella `Esame`. Per prima cosa compilare il file `studente.txt` con le informazioni richieste. In particolare, cognome, nome, matricola, linguaggio in cui si svolge l'esercizio 1 (Java o C).

Nota bene. Per ritirarsi è sufficiente rinominare il file `studente.txt` in `studenteritirato.txt`, **senza** cancellarlo.

Come procedere. Nella cartella `Esame` trovate i) il testo del compito, ii) il file `studente.txt` sopra citato, iii) due sottocartelle `C-aux` e `java-aux`, contenenti il codice di supporto e lo scheletro delle soluzioni per il quesito di programmazione (quesito 1), iv) infine tre file testo in cui scrivere le vostre risposte ai quesiti di teoria. Svolgere il compito nel modo seguente:

- Per il quesito 1, Alla fine la cartella `C-aux` (o java-aux, a seconda del linguaggio usato) dovrà contenere le implementazioni delle soluzioni al quesito 1, ottenute completando i relativi metodi (o le relative funzioni) contenuti nei file forniti dai docenti; si ricorda ancora una volta che la cartella `java-aux` (o `C-aux`) deve trovarsi all'interno della cartella `Esame`.
- Usare invece i file di testo `teoria_2-3.txt`, `teoria_4-5.txt` e `teoria_6-7.txt` forniti con la distribuzione per le risposte ai quesiti di teoria che corrispondono alla numerazione di ogni singolo file. Si prega di indicare chiaramente l'inizio dello svoglimento di ogni quesito a cui si risponde.

Nota bene: È possibile integrare i contenuti di tali file con eventuale materiale cartaceo *unicamente per disegni, grafici, tabelle, calcoli.* Come spiegato sopra, i file da completare sono già forniti con la distribuzione, quindi in generale non è necessario creare nuovi file.

Avviso importante 1. Per svolgere il quesito di programmazione *si consiglia caldamente l'uso di un editor di testo (ad esempio Geany) e la compilazione a riga di comando*, strumenti più che sufficienti per lo svolgimento del compito. La macchina virtuale mette a disposizione diversi ambienti di sviluppo, quali Eclipse e Javabeans. *Gli studenti che li usano lo fanno a proprio rischio.* In particolare, *se ne sconsiglia l'uso qualora non se ne abbia il pieno controllo e certamente se non si è già in grado di sviluppare servendosi unicamente di un editor e della compilazione a riga di comando.* Qualora lo studente intenda comunque usare un ambiente di sviluppo integrato, si raccomanda di controllare che i file vengano effettivamente salvati nella cartella `java-aux` (o `C-aux`, a seconda del linguaggio usato), in quanto vi è il rischio concreto di perdere il proprio lavoro. In generale, file salvati esternamente alla cartella `Esame` andranno persi al termine della prova e quindi non saranno corretti.

Avviso importante 2. Il codice *deve compilare* correttamente (warning possono andare, ma niente errori di compilazione), altrimenti riceverà **0** punti. Si raccomanda quindi di scrivere il programma in modo incrementale, ricompilando il tutto di volta in volta.

Quesito di programmazione

1. [10 points] **Raggiungibilità e distanze minime.** In questo problema si fa riferimento a grafi semplici *diretti*. In particolare, ogni nodo ha associata la lista dei vicini uscenti (vicini ai quali il nodo è connesso da archi uscenti) e dei vicini entranti (nodi che hanno archi diretti verso il nodo considerato). Il grafo è descritto nella classe `Graph.java` (Java) e nel modulo `graph.c` (C). Il generico nodo è descritto nella classe *interna* `Graph.Node` (Java) e in `struct graph_node` (C). Sono inoltre già disponibili le primitive di manipolazione del grafo: creazione di grafo vuoto, lista dei vicini uscenti ed entranti di ciascun nodo (quest'ultima non necessaria per lo svolgimento di questo esercizio), ecc. Per dettagli sulle segnature di tali primitive, così come per dettagli su quali porzioni di memoria allocata vengono liberate dalle varie primitive di cancellazione, si rimanda ai file sorgente distribuiti e ai commenti in essi contenuti. In particolare, per Java si rimanda a `Graph.java` (che contiene la classe `Node` come classe interna). Per C si rimanda all'header `graph.h`. Si noti che le primitive forniscono un insieme base per la manipolazione di grafi, ma i problemi proposti possono essere risolti usandone solo un sottoinsieme.

   Tutto ciò premesso, risolvere al calcolatore quanto segue, in Java o C, con l'avvertenza che gli esempi dati nel testo che segue fanno riferimento alla figura seguente. che corrisponde al grafo usato nel programma di prova (`Driver.java` o `driver`):

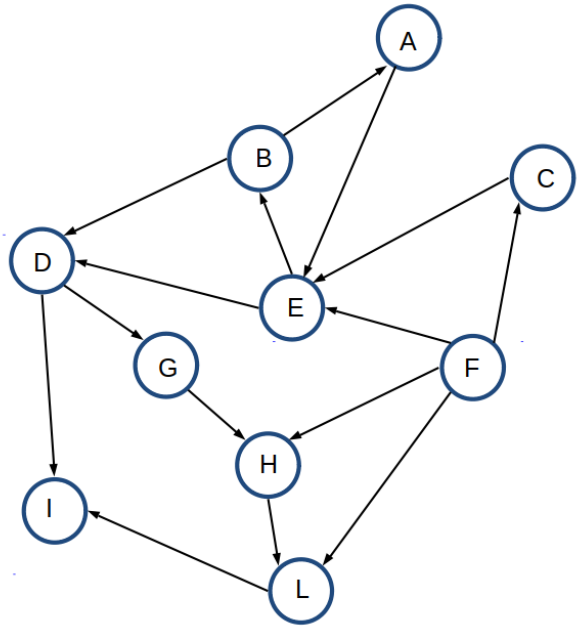

Figure 1: Esempio di grafo diretto non pesato. Le distanze minime dal nodo A sono [A:0, B:2, E:1, D:2, G:3, H: 4, I: 3, L: 5], mentre le distanze minime dal nodo H sono [H:0, L:1, I:2]. Si noti che in entrambi i casi considerati, non tutti i nodi del grafo sono raggiungibili dal nodo di partenza.

   Implementare la funzione/metodo `static <V> void distances(Graph<V> g)` della classe `GraphServices` (o `void distances(graph* g)` del modulo `graph_services` in C) che,

dato un grafo `g` stampa, *per ogni* nodo `source` di `g`, l'elenco (eventualmente vuoto) di tutti i nodi *raggiungibili* da `source` e, per ognuno di questi, la sua *distanza minima* da `source`, intesa come il *numero minimo di archi da attraversare* (l'ordine di stampa non è importante). Per un esempio, si faccia riferimento alla Figura 1.

La valutazione delle risposte degli studenti terrà conto dell'efficienza delle soluzioni proposte.

Nota: Per tenere traccia della distanza dei vari nodi dalla sorgente si consiglia di usare il campo `int timestamp` della classe `Graph.Node` (`int timestamp` di `struct graph_node`). Si consiglia di definire un metodo/funzione ausiliario che, dato `g` e un generico nodo `source`, restituisce (o modifica) una lista contenente i nodi raggiungibili da `source`.

Algoritmi e Strutture Dati - nozioni di base

2. [**No studenti con DSA**] **Algoritmi di Ordinamento.** Si consideri il seguente array **A**:

4	13	7	1	8

   (a) [4 points] Si descriva l'evoluzione dell'algoritmo HEAP-SORT a partire dall'array **A**, mostrando l'albero che rappresenta il max-heap al termine di ogni chiamata di MAX-HEAPIFY.

3. **Alberi Minimi Ricoprenti.** Si consideri il grafo non diretto e pesato in Figura 2

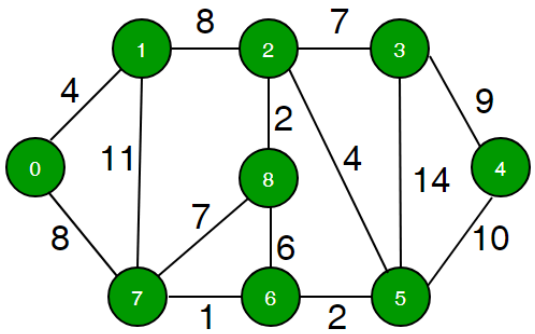

Figure 2: Grafo non diretto pesato.

   (a) [4 points] Si mostri l'evoluzione dell'algoritmo di **Kruskal** per il grafo in Figura 2. In particolare, indicato con T l'insieme degli archi dell'albero minimo ricoprente (inizialmente vuoto), per ogni iterazione i dell'algoritmo occorre specificare i) l'arco che verrà aggiunto a T nell'iterazione i-esima e ii) il peso complessivo dell'albero parziale ottenuto al termine dell'iterazione. Per descrivere l'evoluzione dell'algoritmo usare una tabella con il formato di quella sottostante (o equivalente):

Iterazione	**Arco aggiunto a T**	**Peso complessivo**
1	...	...
2	...	...

Nota bene: descrizioni incomplete o confuse saranno penalizzate.

Algoritmi e Strutture Dati - proprietà di algoritmi e problemi

4. [**No studenti con DSA**] **Componenti fortemente connesse.** Si considerino grafi orientati. Se $G = (V, E)$ è un grafo orientato, la sua versione non orientata $G' = (V, E')$ è definita in modo ovvio trascurando la direzione degli archi di G. In particolare, l'insieme dei vertici è lo stesso. Inoltre, considerati due vertici $a, b \in V$, l'arco *non orientato* $\{a, b\}$ appartiene a E' se e solo se *almeno uno degli* archi *orientati* (a, b) o (b, a) appartiene a E.

   (a) [2 points] Dato un grafo orientato $G = (V, E)$, si definisca formalmente il concetto di componente fortemente connessa di G.

   (b) [2 points] Sia $G' = (V, E')$ la versione non orientata di un generico grafo orientato $G = (V, E)$ come descritto sopra. Siano cc_G e $cc_{G'}$ il numero di componenti connesse di G e G'. *Dimostrare* che $cc_G \geq cc_{G'}$.

   Il punteggio dipenderà molto dal rigore, dalla chiarezza espositiva e dalla solidità delle argomentazioni proposte. Non occorre scrivere tanto ma scrivere bene.

5. **Tabelle hash.** Si consideri una tabella hash T avente m posizioni, nella quale le collisioni sono gestite mediante concatenazione/liste di trabocco. La funzione hash h usata per mappare le chiavi nelle diverse posizioni della tabella si comporta in modo ideale, ossia è *uniforme e indipendente*, come visto a lezione. Indicata con $T[i]$ la i-esima lista di trabocco, un'operazione di ricerca della chiave k comporta la scansione della lista $T[h(k)]$, finché la chiave k non è trovata oppure si raggiunge la fine della lista. Infine, la tabella contiene inizialmente $s \leq cm$ chiavi precedentemente inserite usando la funzione h definita sopra, dove $c > 0$ è una *costante assoluta*.

   (a) [3 points] Si supponga ora che venga eseguita una sequenza di n operazioni di *ricerca* (non sono effettuate operazioni di altro tipo). Calcolare il *valore atteso* asintotico del costo *complessivo* $C(n)$ delle n operazioni di ricerca rispetto alla distribuzione delle s chiavi precedentemente inserite all'interno della tabella.

   Suggerimento: si osservi che date le ipotesi fatte sopra, l'informazione essenziale è che c è una costante.

   Il punteggio dipenderà molto dal rigore, dalla chiarezza espositiva e dalla solidità delle argomentazioni proposte. Non occorre scrivere tanto ma scrivere bene.

Problem Solving e Tecniche Algoritmiche

6. **[No studenti con DSA] Problem solving.** Dato un *intero positivo* n, siamo interessati a $\lfloor \log_2 n \rfloor$, la parte intera del logaritmo in base 2 di n. Ad esempio, $\lfloor \log_2(23) \rfloor = 4$.

   (a) [2 points] Progettare e scrivere lo *pseudo-codice* di un algoritmo *efficiente* `int_log2(n)` che, dato un intero positivo n, restituisce $\lfloor \log_2 n \rfloor$ usando *soltanto addizioni/sottrazioni e divisioni intere (ossia con troncamento).* Mostrare in modo convincente perché l'algoritmo è corretto.

   (b) [2 points] Calcolare la complessità computazionale dell'algoritmo proposto rispetto al parametro n.

   Suggerimento: si osservi che, per $i > 0$, se un numero x cade nell'intervallo $[2^i, 2^{i+1})$, allora $x/2$ (il simbolo "/" indica la divisione con troncamento) cade nell'intervallo $[2^{i-1}, 2^i)$. Ad esempio, $23 \in [2^4, 2^5)$ e $23/2 = 11 \in [2^3, 2^4)$.

   Il punteggio dipenderà molto dal rigore, dalla chiarezza espositiva e dalla solidità delle argomentazioni proposte. Non occorre scrivere tanto ma scrivere bene.

7. **Programmazione dinamica.** Sia dato un insieme S di n interi compresi fra 1 e K; bisogna partizionare l'insieme S in due sottoinsiemi S_1 e S_2 in modo che W_1 e W_2, le somme degli elementi nel sottoinsieme S_1 e S_2 rispettivamente, siano le più vicine possibili fra loro. Equivalentemente si chiede di trovare la partizione che minimizza $|W_1 - W_2|$, il valore assoluto della differenza fra W_1 e W_2.

   (a) [2 points] Scrivere lo pseudo-codice algoritmo efficiente che, dato in input S restituisce una coppia di insiemi S_1e S_2 che rappresentano una partizione di S e verificano la condizione posta.

   (b) [1 point] Calcolare la complessità dell'algoritmo proposto.

   Nota: Si ricorda che S_1 e S_2 sono una partizione di un insieme S se ogni elemento di S è presente o in S_1 o in S_2 ma non in entrambi.