Strutture Dati Algoritmi e Complessità **Name:** _____
A.A. 2023/2024
Compito A
12/07/2024
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] **Distanze minime su grafi pesati.** In questo problema si fa riferimento a grafi semplici diretti, con pesi sugli archi, rappresentati con liste di incidenza. In particolare, il grafo è una lista di coppie (nodo, lista di incidenza). La lista di incidenza di un nodo contiene tutti gli archi uscenti dal nodo, nonché i loro pesi. Più in dettaglio, ogni arco di una lista di incidenza è una terna `(source, target, weight)`, con `source` e `target` rispettivamente i nodi origine e destinazione dell'arco e `weight` il suo peso (positivo). La rappresentazione degli archi è data dalla classe `Edge` (Java) o dalla `struct graph_edge` in `graph.h` . Sono inoltre già disponibili le primitive di manipolazione del grafo (get lista nodi, get lista archi aventi origine da un nodo dato 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. 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. Infine, le classi `MinHeap` e `HeapEntry` (Java) o il modulo `min_heap.c` (in C) implementano un heap minimale che gestisce coppie con chiavi intere e valori di tipo generico. 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`):

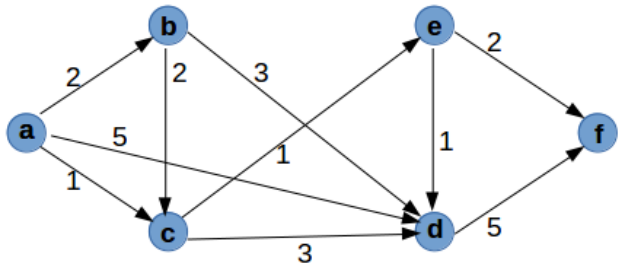

Figure 1: Esempio di grafo diretto pesato. Le distanze minime dal nodo a sono [a:0, b:2, d:3, c:1, e:2, f:4], mentre le distanze minime dal nodo c sono [a:100000, b:100000, d:2, c:0, e:1, f:3], assumendo di rappresentare la distanza infinita attraverso un valore maggiore della somma dei pesi di tutti archi (in questo caso si è scelto il valore 100000)

Implementare la funzione/metodo `public static <V> void distances(Graph<V> g)` della classe `GraphServices` (o `void distances(graph* g)` del modulo `graph_services.c` in C) che, dato un grafo `g`, stampa le distanze minime da *ogni* nodo a tutti gli altri nodi raggiungibili nel grafo. Per i nodi non raggiungibili, la distanza andrà impostata a un valore di riferimento (lo studente può ad esempio usare il valore 100000). Ad esempio, se il grafo fosse quello in figura, la funzione/metodo dovrebbe stampare la seguente stringa o equivalente per il nodo `a` (analogamente per gli altri nodi) `[a:0, b:2, d:3, c:1, e:2, f:4]` . Si faccia anche riferimento alla Figura 1. *Si noti l'ordine di stampa non è importante.*

Suggerimenti: i) In Java/C: si consiglia di definire una funzione/metodo privato che calcola le distanze minime dal generico nodo del grafo; ii) In Java: si consiglia di us-

are una `HashMap` per associare i nodi del grafo a corrispondenti oggetti `HeapEntry` se necessario.

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

Algoritmi e Strutture Dati - nozioni di base

2. [**No studenti con DSA**] **Visita in profondità (DFS).** Si consideri il grafo orientato in Figura 2. *Si assuma che una DFS eseguita su tale grafo esamini i vertici in ordine alfabetico e che ogni lista di adiacenza sia ordinata alfabeticamente.*

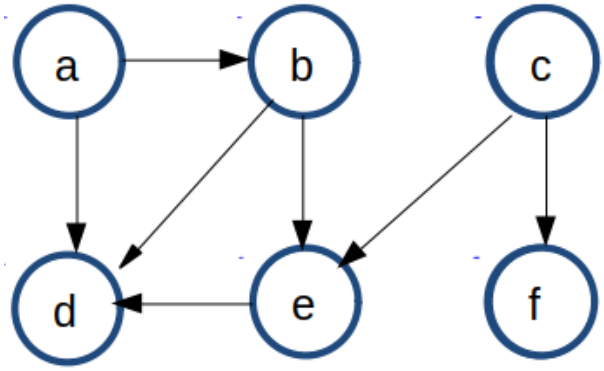

Figure 2: Grafo diretto.

   (a) [4 points] 1) Illustrare la struttura a parentesi della visita in profondità del grafo in Figura 2; 2) indicare la classificazione di ciascun arco (si usi T per denotare gli archi d'albero B per gli archi all'indietro, F per gli archi in avanti e C per gli archi trasversali).

3. **Algoritmi di Ordinamento.** Si consideri il seguente array **A**:

7	2	9	13	15	1	5	3

   (a) [4 points] Si descriva l'evoluzione dell'algoritmo Quick Sort a partire dall'array **A**. Si assuma che, nel processare il generico sotto-array $\mathbf{A}[p...r]$, l'elemento scelto come pivot sia sempre quello di indice più basso, ossia $\mathbf{A}[p]$.

Non è necessario scrivere molto e non è necessario scrivere l'algoritmo, ma occorre descrivere in modo chiaro e non ambiguo l'evoluzione dell'algoritmo.

Algoritmi e Strutture Dati - proprietà di algoritmi e problemi

4. **[No studenti con DSA] Alberi Ricoprenti Minimi.** Sia $G = (V, E)$ un grafo non orientato, connesso e con pesi positivi sugli archi. In particolare, per $(u, v) \in E$, si denoti con $w(u, v)$ il peso dell'arco (u, v). Ciò premesso:
   (a) [2 points] Si definisca *formalmente* il problema di trovare un albero ricoprente minimo in G.
   (b) [2 points] Si *dimostri formalmente* che se (u, v) è un arco di peso *minimo* in G allora esiste almeno un albero ricoprente minimo di G che contiene (u, v).

   Suggerimento: si consiglia di procedere per assurdo.

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

5. **Heap.** Si consideri il problema di costruire un heap a partire da un array. Abbiamo visto a lezione l'algoritmo BUILD-MAX-HEAP, che risolve il problema con costo lineare. Un modo alternativo di costruire un heap è quello ovvio di inserire gli elementi nell'heap uno alla volta. Un algoritmo che fa ciò senza usare array ausiliari è BUILD-MAX-HEAP2($\mathbf{A}$, n), dove $\mathbf{A}$ è l'array da trasformare in heap e n è la sua dimensione.

```
BUILD-MAX-HEAP2(A, n)
Input: Array A[1...n]
A.heapsize = 1
for i = 2 to n do
 |  MAX-HEAP-INSERT(A, A[i], n)
end
```

Algorithm 1: Algoritmo BUILD-MAX-HEAP2.

   Inizialmente, l'heap corrisponde ad $\mathbf{A}[1]$. Alla fine dell'i-esima iterazione, l'heap parziale corrisponde al sotto-array $\mathbf{A}[1 \dots i]$. MAX-HEAP-INSERT($\mathbf{A}$, $\mathbf{A}[i]$, n) è l'algoritmo di inserimento di un nuovo elemento nell'heap visto a lezione. Tra le altre cose, vi ricordiamo che esso incrementa $\mathbf{A}.heapsize$ ogni volta che un nuovo elemento viene inserito.
   (a) [1 point] Dimostrare con un controesempio che BUILD-MAX-HEAP2 e la procedura BUILD-MAX-HEAP vista a lezione non creano necessariamente lo stesso heap se vengono eseguite sullo *stesso* array di input.
   (b) [2 points] Dimostrare che a differenza di BUILD-MAX-HEAP, nel caso peggiore BUILD-MAX-HEAP2 richiede un costo $\Omega(n \log n)$ per costruire un heap di n elementi.

   Suggerimento: Per il secondo punto si può assumere vero il seguente fatto (che dovrebbe esservi noto) senza bisogno di dimostrarlo: un albero binario completo contenente k nodi ha altezza $\lceil \log_2 k \rceil$.

   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.** Si supponga di avere n scatole di dimensioni identiche ma aventi pesi in generale diversi tra loro a causa dei rispettivi contenuti. Sia $\mathbf{A}[i]$ il peso della i-esima scatola, per $i = 1, \ldots, n$. Le scatole vanno caricate su un camion che può trasportare un peso massimo complessivo W. In generale, il peso complessivo delle scatole può eccedere W, per cui solo un sottoinsieme delle scatole può essere caricato.

   (a) [2 points] Progettare e scrivere lo *pseudo-codice* di un algoritmo *efficiente* che, dati l'array $\mathbf{A}$ dei pesi delle scatole e il parametro W, restituisce un sottoinsieme di scatole da caricare nel camion (i rispettivi indici) di cardinalità *massima*.

   Suggerimento: Si pensi ad algoritmi avidi (o Greedy).

   (b) [2 points] Calcolare il costo computazionale dell'algoritmo proposto.

   Il punteggio dipenderà molto dal rigore, dalla chiarezza espositiva e dalla solidità delle argomentazioni proposte. Non occorre scrivere tanto ma scrivere bene. Gli algoritmi devono essere descritti in pseudo-codice e comprensibili (ad esempio: variabili definite ecc.).

7. **Programmazione dinamica.** Dato un array $\mathbf{A}$ di interi positivi di dimensione N, il compito è trovare la lunghezza della sottosequenza *crescente* più lunga (LIS), ovvero la lunghezza della sottosequenza più lunga possibile tra quelle i cui elementi sono ordinati in senso crescente (e non sono necessariamente adiacenti). *Si assuma che i valori di* $\mathbf{A}$ *siano tutti diversi tra loro.* Ad esempio:

   Input: $\mathbf{A} = \{3, 10, 2, 1, 20\}$; Output: 3 (la sottosequenza crescente più lunga è $\{3, 10, 20\}$)

   Input: $\mathbf{A} = \{50, 3, 10, 7, 40, 80\}$; Output: 4 (la sottosequenza crescente più lunga è $\{3, 7, 40, 80\}$)

   Input: $\mathbf{A} = \{2, 1, 20, 4, 5\}$; Output: 3 (sono presenti due sottosequenze crescenti di lunghezza 3)

   (a) [2 points] Scrivere un algoritmo che risolve il problema della sottosequenza crescente più lunga.

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

   Suggerimento: Si suggerisce di attraversare l'array di input dal primo all'ultimo elemento e, per ogni indice i, trovare la lunghezza della sottosequenza crescente più lunga (LIS) che termina esattamente con $\mathbf{A}[i]$; pertanto, dato i, la sequenza include $\mathbf{A}[i]$ e un sottoinsieme degli elementi in $\{\mathbf{A}[1], \mathbf{A}[2], \ldots, \mathbf{A}[i-1]\}$. Sia $L[i]$ la lunghezza trovata per $\mathbf{A}[i]$; alla fine l'output richiesto è il massimo di tutti i valori $L[i]$. Per calcolare $L[i]$ si inizia da $i = 1$ e si procede. il calcolo di $L[i]$ può esser fatto utilizzando l'informazione sulla LIS per tutti gli elementi più piccoli a sinistra.