



Laboratorio n.0

Esercizio n. 1: Sequenze numeriche in vettore

Esercizio da 2 punti, parte di programmazione semplificata, appello del 13 settembre 2018

Sia V un vettore di N interi ($N \leq 30$). Si scriva un programma in C che, una volta acquisito da tastiera tale vettore, visualizzi tutti i sottovettori di dimensione massima formati da celle contigue contenenti dati non nulli.

Esempio:

dato il vettore [1 3 4 0 1 0 9 4 2 0], i due sottovettori di dimensione massima (3) contenenti dati non nulli sono [1 3 4] e [9 4 2].

Esercizio n. 2: Manipolazione di stringhe

Esercizio da 2 punti, parte di programmazione semplificata, appello del 21 giugno 2018

Un file di testo contiene informazioni con il seguente formato:

- la prima riga del file contiene un intero N che indica il numero di parole
- ciascuna delle N righe successive contiene una parola per riga (massimo 20 caratteri).

Si scriva un programma C che conti, tra le parole del file, quante sono le sottostringhe di una data lunghezza con 2 vocali.

Il programma:

- legga i dati da un file di ingresso, il cui nome (massimo 20 caratteri) sia letto da tastiera
- legga da tastiera un intero n che rappresenta la lunghezza delle sottostringhe cercate
- per ogni parola acquisita chiama la funzione `conta` che conta quante sottostringhe di n caratteri contenenti esattamente due vocali appaiono nella stringa S passata come argomento
- al termine stampi il numero complessivo di sottostringhe trovate con esattamente due vocali.

Il prototipo della funzione sia:

```
int conta(char S[20], int n);
```

Esempio:

se $S = "forExample"$ e $n=4$, le sottostringhe di S di lunghezza 4 con 2 vocali sono 4 e sono "forE", "orEx", "rExa" e "Exam".

Esercizio n. 3: Rotazione di vettori

Si scriva una funzione C in grado di permettere all'utente di far ruotare verso destra o verso sinistra i contenuti di un vettore di N interi, di un numero a scelta di posizioni P . Il vettore è da intendersi come *circolare*, nel senso che l'elemento a destra della cella di indice $N-1$ è la cella di indice 0 e l'elemento a sinistra della cella di indice 0 è la cella di indice $N-1$. La figura seguente illustra una rotazione a destra di 3 posizioni:



La funzione abbia il seguente prototipo:

```
void ruota(int v[maxN], int N, int P, int dir);
```

Il main:

1. acquisisca da tastiera N ($N \leq \text{maxN}$ con maxN pari a 30)
2. acquisisca da tastiera il vettore V
3. effettui ripetutamente delle rotazioni, acquisendo ciascuna volta P ($P < N$, $P=0$ per terminare) e la direzione (dir = -1 per rotazione a destra, dir = 1 per rotazione a sinistra) e stampi il vettore risultante.

Esercizio n. 4: Iterazione su matrici

Un file di testo contiene una matrice di interi con il seguente formato:

- la prima riga del file specifica le dimensioni della matrice (numero di righe nr e numero di colonne nc). Si assume che entrambi i valori siano comunque al più pari a 20
- ciascuna delle nr righe successive contiene gli nc valori corrispondenti a una riga della matrice, separati da uno o più spazi.

Si scriva un programma C che:

- legga tale matrice dal file di ingresso, il cui nome (massimo 20 caratteri) sia letto da tastiera
- chieda ripetutamente all'utente un valore dim compreso tra 1 e il minimo tra nr e nc e stampi tutte le sottomatrici quadrate di tale dimensione contenute nella matrice
- termini l'iterazione se l'utente inserisce un valore non coerente con le dimensioni della matrice
- memorizzi in un'opportuna matrice e stampi al termine la sottomatrice quadrata, tra quelle precedentemente individuate, la somma dei cui elementi è massima.



Laboratorio n.1

Esercizio n. 1: Giornate di campionato

Esercizio da 2 punti, parte di programmazione semplificata, appello del 13 settembre 2018

In un campionato n (max 20) squadre giocano per m (max 20) giornate. Sia data una matrice di $n \times m$ numeri interi, ognuno dei quali può valere soltanto 0, 1 o 3. Ogni riga della matrice rappresenta i punti acquisiti dalle n squadre nelle partite disputate nelle m giornate del campionato: 3 punti per le partite vinte, 1 punto per quelle pareggiate e 0 punti per le sconfitte. I risultati della giornata k -esima sono contenuti nelle righe della colonna di indice k . Si scriva un programma C che acquisisca i contenuti di tale matrice da file e che, per ogni giornata del campionato, stampi l'indice (il numero di riga corrispondente) della squadra capolista. Si definisca un opportuno formato per il file.

Esercizio n. 2: Ricodifica di testo con dizionario

Un file (*sorgente.txt*) contiene un testo composto da un numero indefinito di righe, di lunghezza massima 200 caratteri ognuna. Un secondo file (*dizionario.txt*) contiene un elenco di coppie di stringhe. Il file *dizionario.txt* è organizzato come segue:

- sulla prima riga è presente un numero intero e positivo S (≤ 30), che indica il numero di possibili ricodifiche (sostituzioni) presenti nel dizionario
- seguono S coppie <ricodifica><originale> a rappresentare le sostituzioni possibili. Ogni sostituzione <compresso> è nella forma \$<intero>\$

Lo scopo del programma è di ricodificare il primo file di testo andando a sostituire sequenze di caratteri sulla base dei contenuti del secondo file. In caso di più sostituzioni possibili per una certa sottostringa, il programma scelga la prima sostituzione trovata. Il risultato della ricodifica sia salvato su un terzo file (*ricodificato.txt*).

Esempio:

Il contenuto del file *sorgente.txt* è:

apelle figlio di apollo
fece una palla di pelle di pollo
tutti i pesci vennero a galla
per vedere la palla di pelle di pollo
fatta da apelle figlio di apollo

Il contenuto del file *dizionario.txt* è:

9
\$11\$ pelle
\$2\$ pollo
\$333\$ palla
\$41\$ alla
\$5078\$ tta
\$6\$ tti
\$7\$ ll



\$81\$ er
\$900\$ ere

Il file di uscita ricodificato.txt conterrà:

a\$11\$ figlio di a\$2\$
fece una \$333\$ di \$11\$ di \$2\$
tu\$6\$ i pesci venn\$81\$o a g\$41\$
p\$81\$ ved\$900\$ la \$333\$ di \$11\$ di \$2\$
fa\$5078\$ da a\$11\$ figlio di a\$2\$

Esercizio n. 3: Rotazione di matrici

Si scriva un programma C che permetta all'utente di eseguire in sequenza operazioni di rotazione di P posizioni su righe e/o colonne specificate di una matrice di interi. Le rotazioni sono da intendersi come circolari sia sulle righe, sia sulle colonne (cfr. definizione data in Lab. 0 es. 3).

La figura seguente illustra il risultato di alcune operazioni in sequenza:

<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td>9</td></tr></table>	1	2	3	4	5	6	7	8	9	<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>6</td><td>4</td><td>5</td></tr><tr><td>7</td><td>8</td><td>9</td></tr></table>	1	2	3	6	4	5	7	8	9	<table border="1"><tr><td>1</td><td>2</td><td>5</td></tr><tr><td>6</td><td>4</td><td>9</td></tr><tr><td>7</td><td>8</td><td>3</td></tr></table>	1	2	5	6	4	9	7	8	3	<table border="1"><tr><td>2</td><td>5</td><td>1</td></tr><tr><td>6</td><td>4</td><td>9</td></tr><tr><td>7</td><td>8</td><td>3</td></tr></table>	2	5	1	6	4	9	7	8	3	<table border="1"><tr><td>7</td><td>5</td><td>1</td></tr><tr><td>2</td><td>4</td><td>9</td></tr><tr><td>6</td><td>8</td><td>3</td></tr></table>	7	5	1	2	4	9	6	8	3
1	2	3																																															
4	5	6																																															
7	8	9																																															
1	2	3																																															
6	4	5																																															
7	8	9																																															
1	2	5																																															
6	4	9																																															
7	8	3																																															
2	5	1																																															
6	4	9																																															
7	8	3																																															
7	5	1																																															
2	4	9																																															
6	8	3																																															
inizio	riga 2 destra 1 posizione	colonna 3 giù 2 posizioni	riga 1 sinistra 4 posizioni	colonna 1 su 2 posizioni																																													

Il programma:

- legge da file, il cui nome di al massimo 20 caratteri è acquisito da tastiera, la matrice (max 30 x 30). Il formato del file prevede sulla prima riga 2 interi che indicano il numero di righe `nr` e di colonne `nc`, seguita da `nr` righe contenenti ciascuna `nc` interi
- acquisisce ripetutamente da tastiera una stringa (al massimo di 100 caratteri, contenente eventuali spazi), nella forma

`<selettore> <indice> <direzione> <posizioni>`

Il selettore indica se si vuole operare su una riga ("riga"), su una colonna ("colonna"), o terminare ("fine"). Seguono l'indice della riga (colonna) selezionata, la direzione ("destra" o "sinistra", oppure "su" o "giu") e il numero di posizioni.

Le rotazioni siano eseguite da una funzione che generalizza quanto sviluppato per l'esercizio 3 del laboratorio 0.

Valutazione: l'esercizio 2 sarà oggetto di valutazione

Scadenza: caricamento di quanto valutato: entro le 23:59 del 29/10/2019.



Esercitazione di laboratorio n. 3

Esercizio n.1: Individuazione di regioni

Competenze: lettura/scrittura di file, manipolazioni di matrici statiche; puntatori e passaggio di parametri per riferimento (Puntatori e strutture dati dinamiche: 1.4)

Categoria: problemi di verifica e selezione (Dal problema al programma: 4.5)

Si riveda l'esercizio 1 del Lab02 apportandovi le seguenti modifiche:

- supponendo di avere dichiarato una matrice di interi M e di aver definito MAXR come 50 si acquisisca la matrice mediante una funzione (leggiMatrice) che ne ritorna il numero di righe e di colonne effettivamente usati, come parametri “by reference” (o meglio, con puntatori by value). La funzione deve poter essere chiamata con un’istruzione del tipo:

```
leggiMatrice(M, MAXR, &nr, &nc);
```

- per effettuare il riconoscimento delle regioni si utilizzi una funzione riconosciRegione che, data una casella della matrice, determini se si tratti o meno di estremo superiore sinistro di una regione, ritornandone “by reference” (come per la precedente) le dimensioni del rettangolo, e avente come valore di ritorno un intero booleano (vero: rettangolo trovato, falso: rettangolo non trovato). La funzione deve poter essere chiamata come segue:

```
if (riconosciRegione(M, nr, nc, r, c, &b, &h)) {  
    // stampa messaggio per rettangolo con  
    // estremo in (r,c), base b e altezza h  
    ...  
}
```

Esercizio n.2: Puntatori e rappresentazione dati

Competenze: puntatori, codifica dell’informazione, rappresentazioni numeriche

Categoria: il tipo di dato puntatore (Puntatori e strutture dati dinamiche 1.1, 1.2, 1.3)

Si realizzi una funzione che permetta di visualizzare la codifica interna (binaria) di un numero reale, realizzato, in C, da un float, double o long double.

Premessa: i tipi C float, double e long double (se ne veda, ad esempio, la definizione su https://en.wikipedia.org/wiki/C_data_types) realizzano le specifiche IEEE-754 (https://it.wikipedia.org/wiki/IEEE_754) per i tipi di dato reali in precisione singola, doppia ed estesa/tripla/quadrupla. Si noti che per il tipo long double lo standard C non ha una scelta univoca, ma tutti i formati per long double hanno 15 bit di esponente.

Il programma C :

- usa 3 variabili per numeri reali (af, ad, ald, rispettivamente di tipo float, double e long double)
- determina (utilizzando un numero intero, a scelta del programmatore) se il calcolatore utilizza la codifica little endian o big endian e assegna di conseguenza il valore vero o falso (come intero) a una variabile bigEndian
- visualizza (mediante l’operatore C sizeof) la dimensione (espressa in byte e in bit) delle tre variabili af, ad, ald
- acquisisce da tastiera un numero decimale (con virgola ed eventuale esponente in base 10), assegnandolo alle tre variabili af, ad, ald



- mediante la funzione `stampaCodifica` visualizza la rappresentazione interna del numero nelle tre variabili `af`, `ad`, `ald`.

La funzione `stampaCodifica`, avente prototipo:

```
void stampaCodifica (void *p, int size, int bigEndian);
```

va chiamata tre volte, ricevendo come parametri, rispettivamente il puntatore a una della tre variabili (convertito a `void *`) e la dimensione della variabile:

```
stampaCodifica((void *)&af,sizeof(af),bigEndian);
stampaCodifica((void *)&ad,sizeof(ad),bigEndian);
stampaCodifica((void *)&ald,sizeof(ald),bigEndian);
```

La funzione `stampaCodifica`, utilizzando l'aritmetica dei puntatori, la conoscenza del tipo di codifica e la dimensione ricevuta come parametro, deve stampare il bit di segno, i bit di esponente e i bit di mantissa del numero.

Suggerimenti e/o consigli:

si noti che NON si chiede di rappresentare come numeri esponente e mantissa, ma solo di visualizzarne i bit. Pur essendo possibili varie soluzioni, si consiglia di ricavare la codifica binaria, nella funzione `stampaCodifica`, con la strategia che segue:

- non potendo realizzare in C un vettore di bit, si consiglia di leggere/riconoscere il numero ricevuto come vettore di `unsigned char` (ad esempio, un `float` da 32 bit corrisponde a un vettore di 4 `unsigned char`). Si usa l'`unsigned` per limitarsi a numeri senza segno (non negativi). Ogni elemento del vettore va poi decodificato mediante un algoritmo di conversione a binario (riconoscimento dei bit di un numero senza segno). In pratica, il puntatore `p` (di tipo `void *`) andrà internamente assegnato a un puntatore a `unsigned char`
- occorre stampare i bit, separando bit di segno, di esponente e di mantissa, a partire dal più significativo (MSB). A seconda del parametro `bigEndian`, si stabilisce la direzione in cui percorrere i byte del numero. Il parametro `size` serve per sapere dove terminano i bit di esponente ed iniziano quelli della mantissa. Il percorso sui byte può essere realizzato mediante opportuno utilizzo dell'aritmetica dei puntatori, tale da percorrere tutti i byte del dato dal più significativo al meno significativo (o viceversa, a seconda della scelta fatta per la decodifica).

Valutazione: entrambi gli esercizi 1 e 2 saranno oggetto di valutazione

Scadenza: caricamento di quanto valutato: entro le 23:59 del 29/10/2019.



Esercitazione di laboratorio n. 4

Esercizio n.1: Massimo Comun Divisore

Competenze: ricorsione matematica (Ricorsione e problem-solving: 2.2)

Dati 2 interi positivi a e b , il loro massimo comun divisore ($\text{gcd}(a, b)$) si definisce ricorsivamente come:

$$\text{gcd}(a, b) = \begin{cases} 2 \cdot \text{gcd}\left(\frac{a}{2}, \frac{b}{2}\right) & \text{if } a, b \text{ are even} \\ \text{gcd}\left(a, \frac{b}{2}\right) & \text{if } a \text{ is odd, } b \text{ is even} \\ \text{gcd}\left(\frac{a-b}{2}, b\right) & \text{if } a, b \text{ are odd} \end{cases}$$

Si individui un'opportuna condizione di terminazione e si scriva una funzione ricorsiva `int gcd(int a, int b);` che realizzi la definizione di cui sopra. Si ricordi che nel calcolo del massimo comun divisore si assume che $a > b$. Per tener conto di tale condizione, se non soddisfatta, si scambiano a e b .

Esercizio n.2: Elemento maggioritario

Competenze: ricorsione matematica (Ricorsione e problem-solving: 2.2)

Sia dato un vettore `vet` di N naturali. Si definisce elemento maggioritario, se esiste, quel valore che ha numero di occorrenze $> N/2$.

Esempio: se $N=7$ e `vet` contiene 3, 3, 9, 4, 3, 5, 3 l'elemento maggioritario è 3. Se $N=8$ e `vet` contiene 0, 1, 0, 2, 3, 4, 0, 5 non esiste elemento maggioritario.

Si scriva una funzione `maggioritario` che, dati N e `vet`, visualizzi l'elemento maggioritario se esiste, -1 se non esiste. Il prototipo sia:

```
int majority( int *a, int N );
```

Vincoli: complessità $O(n \log n)$, algoritmo in loco.

Esercizio n.3: Valutazione di espressioni regolari

I problemi di ricerca di stringhe all'interno di testi e/o collezioni di stringhe (solitamente di dimensione maggiore rispetto alla stringa cercata) si basano raramente su un confronto esatto, ma molto spesso necessitano di rappresentare in modo compatto non una, ma un insieme di stringhe cercate, evitandone per quanto possibile l'enumerazione esplicita. Le **espressioni regolari** sono una notazione molto utilizzata per rappresentare (in modo compatto) insiemi di stringhe correlate tra loro (ad esempio aventi una parte comune).

Una **espressione regolare** (o regexp) è una sequenza di simboli (quindi una stringa) che identifica un insieme di stringhe. Si scriva una funzione in C in grado di individuare (cercare) eventuali occorrenze di una data regexp all'interno di una stringa di input.

La funzione sia caratterizzata dal seguente prototipo:

```
char *cercaRegexp( char *src, char *regexp );
```

dove :



- il parametro `src` rappresenta la stringa sorgente in cui cercare.
- il parametro `regexp` rappresenta l'espressione regolare da cercare.
- il valore di ritorno della funzione è un puntatore alla prima occorrenza di `regexp` in `src` (NULL se non trovata).

Ai fini dell'esercizio si consideri di valutare solamente stringhe composte da caratteri alfabetici. Si considerino inoltre solamente espressioni regolari composte da caratteri alfabetici e dai seguenti metacaratteri:

- . trova un singolo carattere (cioè qualunque carattere può corrispondere a un punto)
- [] trova un singolo carattere contenuto nelle parentesi (cioè uno qualsiasi dei caratteri tra parentesi va bene)
- [^] trova ogni singolo carattere non contenuto nelle parentesi (cioè tutti i caratteri tra parentesi non vanno bene)
- \a trova un carattere minuscolo
- \A trova un carattere maiuscolo

Esempi di espressioni regolari:

.oto corrisponde a ogni stringa di quattro caratteri terminante con "oto", es. "voto", "noto", "foto", ...

[mn]oto rappresenta solamente "moto" e "noto"

[^f]oto rappresenta tutte le stringhe terminanti in "oto" ad eccezione di "foto"

\aoto rappresenta ogni stringa di quattro caratteri (come "voto", "noto", "foto", ...) iniziante per lettere minuscola e terminante in "oto"

\Aoto rappresenta ogni stringa di quattro caratteri (come "Voto", "Noto", "Foto", ...) iniziante per lettere maiuscola e terminante in "oto".

NOTA: i metacaratteri possono apparire in qualsiasi punto dell'espressione regolare. I casi qui sopra sono solo una minima parte a titolo di esempio. Sono quindi espressioni regolari valide: A[^f]\anR.d, \A[aeiou]5t[123], ecc.

Esercizio n.4: Azienda di trasporti - ordinamento

Si consideri lo scenario introdotto nell'esercizio 2 del Laboratorio 2. Si realizzi un programma in C che, una volta acquisite le informazioni in una opportuna struttura dati, renda disponibili le seguenti operazioni:

- stampa, a scelta se a video o su file, dei contenuti del log
- ordinamento del vettore per data, e a parità di date per ora
- ordinamento del vettore per codice di tratta
- ordinamento del vettore per stazione di partenza
- ordinamento del vettore per stazione di arrivo
- ricerca di una tratta per stazione di partenza (anche parziale).

Per quanto riguarda le ricerche, si richiede che siano implementate sia una funzione di ricerca dicotomica sia una funzione di ricerca lineare. Per quanto riguarda l'ordinamento, si presti attenzione alla stabilità dell'algoritmo prescelto nel caso di ordinamento per più chiavi successive. Si selezioni l'algoritmo di ricerca più opportuno: se la base dei dati è ordinata secondo la chiave di ricerca corrente si usi la ricerca dicotomica, altrimenti quella lineare. Si suggerisce a tal proposito di



mantenere nel programma uno stato relativo all'ordinamento corrente della base dati (ossia, su quale chiave sia attualmente ordinato).

Esercizio n.5: Azienda di trasporti - multiordinamento

A partire dalle specifiche dell'esercizio precedente, estendere le funzionalità del programma per mantenere in contemporanea più ordinamenti della base dati

Suggerimento: si legga il vettore originale una sola volta, mantenendolo nell'esatto ordine di lettura per tutta la durata dell'esecuzione. Si affianchino al vettore originale tanti vettori di puntatori a struttura quanti sono gli ordinamenti richiesti, con i quali sono gestiti gli ordinamenti stessi.

Valutazione: gli esercizi 2, 3 e 5 saranno oggetto di valutazione

Scadenza: caricamento di quanto valutato: entro le 23:59 del 26/11/2019.



Esercitazione di laboratorio n. 5

Esercizio n.1: Playlist

Competenze: esplorazione dello spazio delle soluzioni con i modelli del Calcolo Combinatorio (Ricorsione e problem-solving: 3.2, 3.3)

Un gruppo di amici vuole preparare una playlist per un viaggio. Per accontentare tutti, ogni amico ha diritto a proporre fino a cinque canzoni tra cui scegliere. La playlist viene creata scegliendo tante canzoni quanti sono gli amici, e per ognuno di essi scegliendo una ed una sola canzone tra quelle da lui proposte. Tutte le canzoni sono distinte.

Le informazioni sulle canzoni proposte sono memorizzate in un file testuale (`brani.txt`) organizzato come segue:

- sulla prima riga appare il numero A di amici
- seguono A blocchi di righe, uno per ogni i -esimo amico, tali per cui
 - sulla prima riga del blocco appare il numero C_i di canzoni proposte dall'amico
 - seguono C_i stringhe, in ragione di una per riga, riportanti i titoli dei brani proposti

Si assume che tutti i titoli delle canzoni non contengano spazi e siano lunghi al più 255 caratteri.
Si scriva un programma in C che, letto il file di input, generi tutte le playlist possibili con le regole di cui sopra.

Nota: i contenuti del file di input di esempio in allegato sono indentati per rendere più facile distinguere le porzioni dedicate ad ogni amico.

Si consiglia di individuare preventivamente a quale modello del Calcolo Combinatorio, tra quelli visti a lezione, faccia riferimento il problema e in seguito di adattare il codice presentato in aula.

Esercizio n.2: Allocazione di matrici

Competenze: Strutture dati dinamiche, matrici dinamiche create da funzioni (Puntatori e strutture dati dinamiche: 3.3.3)

Un file di testo contiene una matrice di interi con il seguente formato:

- la prima riga del file specifica le dimensioni della matrice (numero di righe nr e numero di colonne nc)
- ciascuna delle nr righe successive contiene gli nc valori corrispondenti a una riga della matrice, separati da uno o più spazi.

Si scriva un programma che allochi dinamicamente la matrice ed effettui la lettura del file. La funzione di allocazione della matrice può:

- far uso del valore di ritorno per restituire il puntatore alla matrice al `main`
`int **malloc2dR(...);`
- restituire il puntatore alla matrice tra i parametri passati per riferimento
`void malloc2dP(int***, ...);`

Una volta acquisita la matrice, il programma deve invocare una funzione

`void separa(int **mat, int nr, int nc, ...)`

che:

- interpreti la matrice come se fosse una scacchiera (su ogni riga e su ogni colonna celle bianche e nere alternate)



- separi i dati delle caselle nere da quelli delle caselle bianche, copiandoli (in ordine arbitrario) in due vettori dinamici, che vanno ritornati al programma chiamante
- i due vettori di interi, di lunghezza opportuna e tale da poter contenere gli elementi delle "celle bianche" e delle "celle nere" separatamente, vanno allocati dinamicamente
- i due vettori e i loro contenuti devono essere visibili al chiamante della funzione `separa()` (andranno quindi dichiarati in modo opportuno, al posto dei puntini ...), che li stampa e li dealloca.

Esercizio n. 3: Azienda di trasporti - multiordinamento

Competenze: vettori di struct, strutture dati dinamiche, vettori dinamici, (Puntatori e strutture dati dinamiche 2.5.5, 3.3.3, 3.2.3)

Si ripeta l'esercizio 5 del laboratorio 4 utilizzando vettori allocati dinamicamente. In particolare, i vettori vanno allocati in base al numero di dati effettivamente presenti nel file acquisito.

Oltre ai comandi già previsti, si chiede di aggiungerne uno di acquisizione (lettura) di un nuovo file (dato il nome). Si prevede quindi non solo di acquisire una volta sola i dati da file, all'inizio dell'esecuzione, ma di poterlo fare ogni volta che l'utente lo richieda. Il comando di lettura, in generale, prevede l'acquisizione di nuovi dati da un file diverso dal precedente. Occorre quindi liberare (utilizzando la funzione `free`) i vettori precedentemente allocati, e allocarne di nuovi per i nuovi dati da leggere. Si noti che il compito di acquisizione dei dati può essere svolto da una sola funzione, chiamata sia (una prima volta) all'avvio del programma che ad ogni attivazione del comando di lettura.

Valutazione: tutti gli esercizi saranno oggetto di valutazione

Scadenza: caricamento di quanto valutato: entro le 23:59 del 26/11/2019.



Esercitazione di laboratorio n. 6

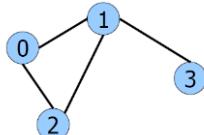
Esercizio n.1: Vertex cover

Competenze: esplorazione dello spazio delle soluzioni con i modelli del Calcolo Combinatorio (Ricorsione e problem-solving: 3.2, 3.3)

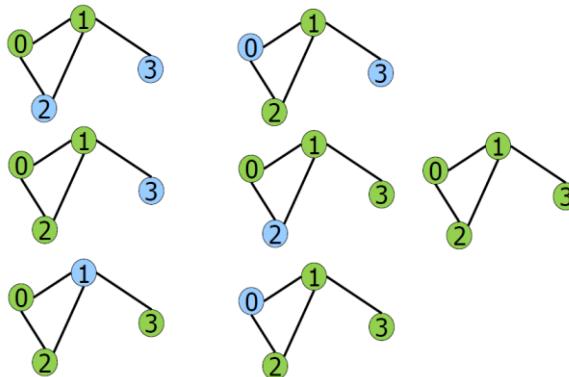
Sia dato un grafo non orientato G di N vertici, identificati da interi nell'intervallo $0..N-1$, ed E archi, identificati come coppie di vertici. Il grafo è memorizzato su di un file, nella cui prima riga compaiono N ed E , mentre nelle E righe successive compaiono, uno per riga, gli archi nella forma $u\ v$.

Un **vertex cover** è un sottoinsieme W dei vertici tali che per tutti gli archi $(u,v) \in E$ o $u \in W$ o $v \in W$. Dopo aver letto da file il grafo G ed aver memorizzato le informazioni rilevanti in opportune strutture dati, si elenchino tutti i vertex cover.

Esempio: per il grafo seguente



esistono i seguenti vertex cover: $(0, 1)$, $(1, 2)$, $(0, 1, 2)$, $(0, 1, 3)$, $(0, 2, 3)$, $(1, 2, 3)$, $(0, 1, 2, 3)$.



Si osservi che questo esercizio non richiede conoscenze di Teoria dei Grafi.

Esercizio n. 2: Anagrafica con liste

Competenze: creazione e gestione di liste concatenate (Puntatori e strutture dati dinamiche: 4.1)

I dettagli di una anagrafica sono memorizzati in file di testo composti da un numero indefinito di righe nella seguente forma:

<codice> <nome> <cognome> <data_di_nascita> <via> <citta'> <cap>

Il campo <data_di_nascita> è nella forma gg/mm/aaaa, <cap> è un numero intero, mentre tutti i campi rimanenti sono stringhe senza spazi di massimo 50 caratteri. <codice> è nella forma AXXXX, dove X rappresenta una cifra nell'intervallo 0-9, ed è univoco nell'intera anagrafica. I dettagli dell'anagrafica vanno racchiusi in un opportuno tipo di dato Item.

L'anagrafica va memorizzata in una lista ordinata per data di nascita (le persone più giovani appaiono prima nella lista).



Si scriva un programma in C che, una volta inizializzata una lista vuota, offra le seguenti funzionalità:

- acquisizione ed inserimento ordinato di un nuovo elemento in lista (da tastiera)
- acquisizione ed inserimento ordinato di nuovi elementi in lista (da file)
- ricerca, per codice, di un elemento
- cancellazione (con estrazione del dato) di un elemento dalla lista, previa ricerca per codice
- cancellazione (con estrazione del dato) di tutti gli elementi con date comprese tra 2 date lette da tastiera. Si consiglia, anziché di realizzare una funzione che cancelli dalla lista questi elementi, restituendoli memorizzati in una lista o in un vettore dinamico, di implementare una funzione che estragga e restituisca al programma chiamante il primo degli elementi appartenenti all'intervallo. Il programma chiamante itererà la chiamata di questa funzione, stampando il risultato, per tutti gli elementi dell'intervallo
- stampa della lista su file.

Per le funzioni di ricerca e cancellazione è richiesto che la funzione che opera sulle liste ritorni l'elemento trovato o cancellato al programma chiamante, che provvede alla stampa.

Valutazione: entrambi gli esercizi 1 e 2 saranno oggetto di valutazione

Scadenza: caricamento di quanto valutato: entro le 23:59 del 26/11/2019.



Esercitazione di laboratorio n. 7

Esercizio n.1: Collane e pietre preziose

Competenze: esplorazione dello spazio delle soluzioni con i modelli del Calcolo Combinatorio (Ricorsione e problem-solving: 3.2.4, 3.3.6), problemi di ottimizzazione (Ricorsione e problem-solving: 3.5)

Un gioielliere ha a disposizione z zaffiri, s smeraldi, r rubini e t topazi per creare una collana infilando una pietra dopo l'altra. Deve però soddisfare le seguenti regole:

- uno zaffiro deve essere seguito immediatamente o da un altro zaffiro o da un rubino
- uno smeraldo deve essere seguito immediatamente o da un altro smeraldo o da un topazio
- un rubino deve essere seguito immediatamente o da uno smeraldo o da un topazio
- un topazio deve essere seguito immediatamente o da uno zaffiro o da un rubino.

Si scriva una funzione C che calcoli la lunghezza e visualizzi la composizione di una collana a lunghezza massima che rispetti le regole di cui sopra. La lunghezza della collana è il numero di pietre preziose che la compongono.

Osservazione: la lunghezza della soluzione non è nota a priori, ma può variare tra 1 e $(z+r+s+t)$.

Suggerimento: l'esercizio può essere risolto adottando un approccio simile a quello delle disposizioni con ripetizione, visto a lezione, se opportunamente adattato ai requisiti del problema. Una volta impostato il modello ricorsivo, si scriva poi la funzione di filtro (verifica di accettabilità) e di ottimizzazione. Si valuti infine la possibilità di introdurre criteri di pruning.

Nota: si presti attenzione al crescere del tempo di esecuzione richiesto dall'identificazione della soluzione a fronte dell'aumentare dei valori dei parametri di ingresso per il problema (numero di pietre preziose disponibili).

Esercizio n.2: Collane e pietre preziose (versione 2)

Si consideri il contesto introdotto dall'esercizio precedente. Ogni tipologia di pietra è caratterizzata dal suo valore (intero non negativo) (val_z , val_s , val_r , val_t). Il valore della collana è dato dalla somma dei valori delle singole pietre che la compongono. Indicando con n_z , n_s , n_r e n_t il numero di zaffiri, smeraldi, rubini e topazi, il valore della collana è:

$$\text{val} = \text{val_z} * n_z + \text{val_s} * n_s + \text{val_r} * n_r + \text{val_t} * n_t$$

La composizione della collana deve rispettare tutte le regole introdotte nell'esercizio precedente. In aggiunta, devono essere rispettati i seguenti criteri:

- nessuna tipologia di pietra si può ripetere più di max_rip volte consecutive
- nella collana, il numero di zaffiri non può superare il numero di smeraldi

Si scriva una funzione C che calcoli la composizione di una collana a valore massimo che rispetti le regole di cui sopra.

Esercizio n. 3: Gioco di ruolo

Competenze: Vettori di struct, strutture dati dinamiche, vettori dinamici, riallocazione dinamica (Puntatori e strutture dati dinamiche 2.5.5, 3.3.3, 3.2.3)



Sia dato un file di testo (`pg.txt`), contenente i dettagli di alcuni personaggi di un gioco di ruolo, organizzato come segue:

- il numero di personaggi presenti nel file non è noto a priori
- i dettagli di ogni personaggio sono riportati in ragione di uno per riga. In ogni riga sono presenti tre stringhe quali un codice identificativo univoco, il nome del personaggio e la sua classe. Segue sulla stessa una sestupla a rappresentare le statistiche di base del personaggio, nella forma `<hp> <mp> <atk> <def> <mag> <spr>`. Ai fini dell'esercizio non è rilevante conoscere il significato dei campi della sestupla,
- il codice è nella forma PGXXXX, dove X rappresenta una cifra nell'intervallo 0-9
- il nome e la classe di ogni personaggio sono rappresentati da una stringa, priva di spazi, di massimo 50 caratteri alfabetici (maiuscoli o minuscoli)
- le statistiche sono dei numeri interi positivi o nulli
- tutti i campi sono separati da uno o più spazi.

In un secondo file di testo (`inventario.txt`), sono memorizzati i dettagli di una serie di oggetti a cui i personaggi del gioco hanno accesso. Il file è organizzato come segue:

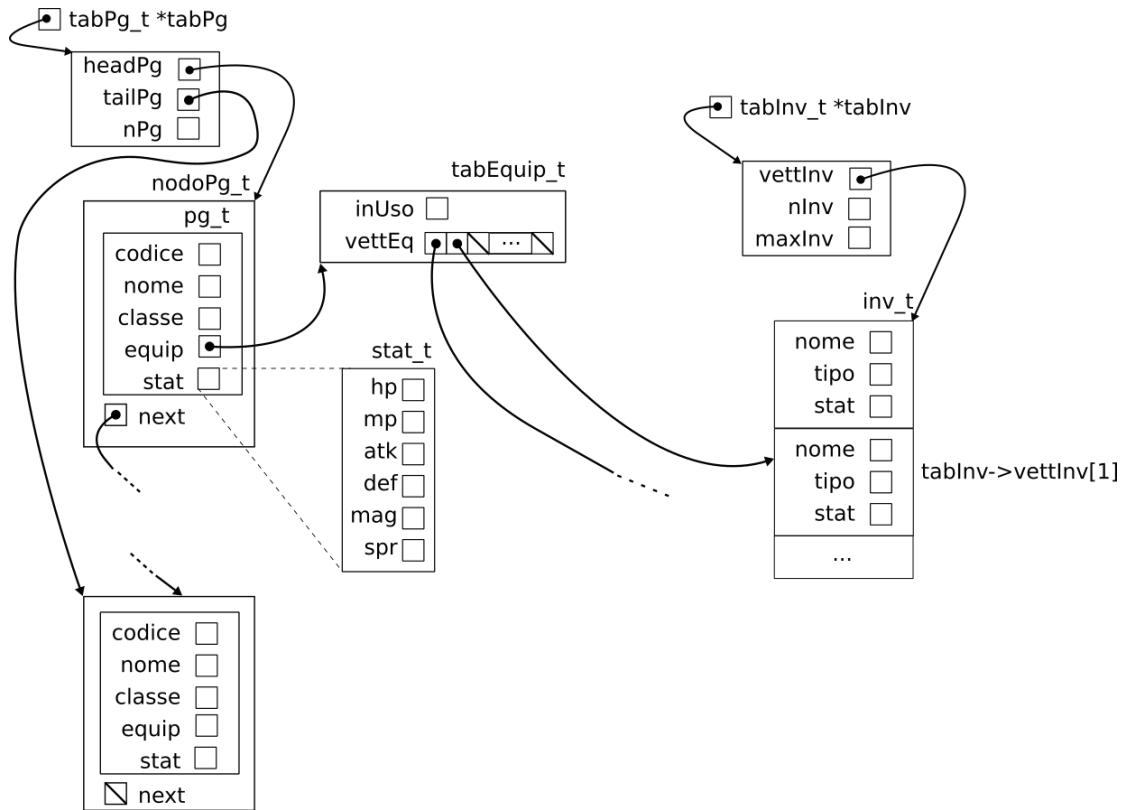
- sulla prima è presente il numero O di oggetti
- sulle O righe successive appaiono i dettagli di ogni oggetto disponibile
- ogni oggetto è caratterizzato da un nome, una tipologia e da una sestupla a rappresentare i modificatori alle statistiche base di un personaggio
- il nome e la tipologia sono rappresentati da una stringa, priva di spazi, di massimo 50 caratteri alfabetici (maiuscoli o minuscoli)
- i modificatori alle statistiche sono dei numeri interi (potenzialmente anche negativi), quindi possono essere visti come *bonus* (se positivi) o *malus* (se negativi).

Ogni personaggio può fare uso degli oggetti disponibili nell'inventario e comporre liberamente il proprio equipaggiamento, fino ad un massimo di otto elementi.

Ai fini dell'esercizio, si imposti la struttura dati in modo che sia coerente con la rappresentazione grafica proposta in figura (i nomi dei tipi e dei campi sono solo a titolo di esempio). Se necessario, si aggiungano tutti i campi addizionali ritenuti opportuni. Si presti particolare attenzione all'uso di strutture *wrapper* per la memorizzazione delle collezioni (per personaggi, oggetti e equipaggiamenti).

Si scriva un programma in C che permetta di:

- caricare in una lista l'elenco di personaggi
- caricare in un vettore di strutture, allocato dinamicamente, l'elenco di oggetti
- aggiungere un nuovo personaggio
- eliminare un personaggio
- aggiungere/rimuovere un oggetto dall'equipaggiamento di un personaggio
- calcolare le statistiche di un personaggio tenendo in considerazione i suoi parametri base e l'equipaggiamento corrente. Si presti attenzione al fatto che nessuna statistica può risultare minore di 1, indipendentemente dagli eventuali *malus* cumulativi dovuti alla scelta di equipaggiamento.



Valutazione: gli esercizi 2 e 3 saranno oggetto di valutazione

Scadenza: caricamento di quanto valutato: entro le 23:59 del 20/12/2019.



Esercitazione di laboratorio n. 8

Esercizio n.1: Sequenza di attività

Competenze: esplorazione dello spazio delle soluzioni con i modelli del Calcolo Combinatorio (Ricorsione e problem-solving: 3.2.4, 3.3.6), problemi di ottimizzazione (Ricorsione e problem-solving: 3.5)

Traccia da 12 punti, esercizio da 6 punti, appello del 29/01/2018

Un’attività i è caratterizzata da un intervallo aperto (s_i, f_i) , dove s_i è il tempo di inizio, f_i è un tempo di fine e $d_i = f_i - s_i$ è la durata dell’attività. Tempo di inizio, di fine e durata sono interi. Una collezione di attività S è memorizzata in un vettore v di strutture `att` aventi come campi tempo di inizio e tempo di fine. Si suppongano corretti i dati contenuti nel vettore ($\forall i \ s_i \leq f_i$). Due attività i e j sono incompatibili se e solo se si intersecano o sovrappongono:

$$(s_i < f_j) \&\& (s_j < f_i) \Leftrightarrow i \cap j \neq \emptyset$$

Si scrivano la funzione wrapper e una funzione ricorsiva in C in grado di determinare e visualizzare un sottoinsieme di attività compatibili che massimizza la somma delle durate:

$$\max \sum_{k \in S} d_k \&\& \forall (k_1, k_2) \in S, k_1 \cap k_2 = \emptyset$$

Il prototipo della funzione wrapper sia:

```
void attSel(int N, att *v);
```

Esempio:

se $S = \{(1,2), (2,4), (2,5), (3,5), (5,7), (6,8)\}$, uno dei sottoinsiemi di S di attività compatibili che massimizza la somma delle durate è $(1,2), (2,5), (6,8)$ per una somma delle durate pari a 6.

Esercizio n.2: Tessere e scacchiere

Competenze: esplorazione dello spazio delle soluzioni con i modelli del Calcolo Combinatorio (Ricorsione e problem-solving: 3.2.4, 3.3.6), problemi di ottimizzazione (Ricorsione e problem-solving: 3.5)

Traccia da 18 punti, appello del 21/06/2018

Un gioco si svolge su una scacchiera rettangolare di $R \times C$ caselle. In ogni casella deve essere posizionata una tessera su cui sono disegnati due segmenti di tubo, uno in orizzontale e uno in verticale. Le diagonali non sono contemplate. Ogni segmento è caratterizzato da un colore e da un punteggio (valore intero positivo).

Per ottenere i punti associati ai vari segmenti, è necessario allineare lungo un’intera riga (colonna) tubi in orizzontale (verticale) dello stesso colore. Le tessere possono essere ruotate di 90° . Le tessere sono disponibili in copia singola. Si assuma esistano abbastanza tessere per completare la scacchiera.

Lo scopo del gioco è ottenere il massimo punteggio possibile posizionando una tessera in ogni cella della scacchiera.



Su un primo file di testo (`tiles.txt`) è riportato l'elenco delle tessere disponibili nel seguente formato:

- sulla prima riga è presente il numero T di tessere
- seguono T quadruple nella forma <coloreT1><valoreT1><coloreT2><valoreT2> a descrivere la coppia di tubi presenti sulla tessera in termini di rispettivo colore e valore.

Si assume che a ogni tessera sia associato un identificativo numerico nell'intervallo 0..T-1

Su un secondo file di testo (`board.txt`) è riportata una configurazione iniziale per la scacchiera di gioco. Il file ha il seguente formato:

- sulla prima riga è presente una coppia di interi R C a rappresentare il numero di righe e colonne della superficie di gioco
- seguono R righe riportanti C elementi ciascuna a definire la configurazione di ogni cella
- ogni cella è descritta da una coppia t_i/r dove t_i è l'indice di una tessera tra quelle presenti in `tiles.txt` e r rappresenta l'eventuale sua rotazione (es: 7/0 oppure 3/1 per rappresentare rispettivamente una tessera non ruotata e una ruotata). Una cella vuota è rappresentata dalla coppia -1/-1.

Si scriva un programma in C che, una volta acquisiti in opportune strutture dati l'elenco delle tessere disponibili e la configurazione iniziale della scacchiera generi la soluzione a punteggio massimo possibili a partire dalla configurazione iniziale letta da file.

Esempio:

<code>tiles.txt</code>	<code>board.txt</code>	scacchiera iniziale	scacchiera finale
9 A 3 B 2 A 2 V 1 A 2 V 2 B 1 N 2 A 3 G 3 V 1 G 2 R 1 G 6 V 1 B 1 V 11 B 3	3 3 0/0 -1/-1 2/0 3/1 -1/-1 -1/-1 -1/-1 6/0 -1/-1		 = 8

Esercizio n.3: Gioco di ruolo (multifile)

Competenze: Vettori di struct, strutture dati dinamiche, vettori dinamici, riallocazione dinamica (Puntatori e strutture dati dinamiche 2.5.5, 3.3.3, 3.2.3). Programmazione multifile(Puntatori e strutture dati dinamiche 5.5)

Si consideri lo scenario introdotto nell'esercizio n. 3 del laboratorio 7 (Gioco di ruolo). Si organizzi il codice scritto precedentemente suddividendolo in più moduli, quali:

- un modulo client contenente il main e l'interfaccia utente/menu
- un modulo per la gestione dei personaggi
- un modulo per la gestione dell'inventario.



Il modulo per i personaggi deve fornire le funzionalità di:

- acquisizione da file delle informazioni dei personaggi, mantenendo la medesima struttura a lista richiesta nel laboratorio precedente
- inserimento/cancellazione di un personaggio
- ricerca per codice di un personaggio
- stampa dei dettagli di un personaggio e del relativo equipaggiamento, se presente
- modifica dell'equipaggiamento di un personaggio
 - aggiunta/rimozione di un oggetto.

Il modulo dell'inventario deve fornire le funzionalità di:

- acquisizione da file delle informazioni relative agli oggetti disponibili, mantenendo la medesima struttura a vettore richiesta nel laboratorio precedente
- ricerca di un oggetto per nome
- stampa dei dettagli di un oggetto.

NB: il modulo personaggi è client del modulo inventario, in quanto ogni personaggio ha una collezione di (riferimenti a) dati contenuti nell'inventario.

Valutazione: gli esercizi 2 e 3 saranno oggetto di valutazione

Scadenza: caricamento di quanto valutato: entro le 23:59 del 20/12/2019.



Esercitazione di laboratorio n. 9

Esercizio svolto n. 0: antenne della rete di telefonia mobile

È dato un insieme di n città, disposte su una strada rettilinea, identificate con gli interi da 1 a n , ognuna caratterizzata dal numero di abitanti (migliaia, intero). In ogni città si può installare un'antenna della rete di telefonia mobile alla sola condizione che le città adiacenti non abbiano l'antenna. Ogni antenna copre solo la popolazione della città dove è posta.

antenne possibili							
città	1	2	3	4	5	6	7
abitanti	14	22	13	25	30	11	90

Con il paradigma della programmazione dinamica bottom-up, determinare il massimo numero di abitanti copribile rispettando la regola di installazione e la corrispondente disposizione delle antenne. Visti i vincoli, è evidente che non si potrà coprire tutta la popolazione di tutte le città

Svolgimento:

si tratta di un problema di ottimizzazione che potrebbe essere risolto identificando tutti i sottoinsiemi di antenne, valutando quelli che soddisfano la regola e, tra questi, quello ottimo. Il modello è quello del powerset.

In alternativa si propone una soluzione basata sul paradigma della **programmazione dinamica**. I dati sono memorizzati in un vettore di interi val di $n+1$ celle. La cella di indice 0 corrisponde alla città fittizia che non esiste e che non ha abitanti.

val	0	10	20	5	15	55	30
	0	1	2	3	4	5	6

Passo 1: applicabilità della programmazione dinamica

Si osservi che è vera la seguente affermazione: la soluzione ottima del problema per la città di indice k corrisponde a uno dei seguenti 2 casi

- nella città k non c'è un'antenna: la soluzione ottima coincide con quella per le prime $k-1$ città
- nella città k c'è un'antenna: la soluzione ottima si ottiene dalla soluzione ottima per le prime $k-2$ città cui si aggiunge l'antenna nella città k .

Supponiamo di memorizzare in un vettore di interi opt di $n+1$ celle scandito da un indice k la soluzione ottima che si ottiene considerando le prime k antenne: $opt[0]=0$ in quanto non ci sono né città, né abitanti, né antenne; $opt[1]=val[1]$ in quanto c'è solamente la città di indice $i=1$ con i suoi abitanti $val[1]$. Per gli altri casi $1 < k \leq n$:

- è possibile piazzare un'antenna nella città k , quindi non è più possibile piazzare un'antenna nella città $k-1$ che la precede, bensì nella città ancora prima $k-2$: $opt[k] = opt[k-2] + val[k]$



- non è possibile piazzare un'antenna nella città k , quindi è possibile piazzare un'antenna nella città $k-1$ che la precede: $\text{opt}[k] = \text{opt}[k-1]$.

Il problema per la città k -esima richiede la soluzione dei sottoproblemi per le città $(k-1)$ -esima o $(k-2)$ -esima. Se $\text{opt}[k-1]$ o $\text{opt}[k-2]$ non fossero massimi, si potrebbero trovare soluzioni $\text{opt}'[k-1] > \text{opt}[k-1]$ o $\text{opt}'[k-2] > \text{opt}[k-2]$ che contraddirebbero l'ipotesi di $\text{opt}[k]$ massimo. La programmazione dinamica è quindi applicabile.

Passo 2: soluzione ricorsiva

L'analisi precedente può essere riassunta con la seguente formulazione ricorsiva:

$$\text{opt}(k) = \begin{cases} 0 & k = 0 \\ 1 & = 1 \\ \max(\text{opt}(k-1), \text{val}(k) + \text{opt}(k-2)) & 1 < k \leq n \end{cases}$$

facilmente codificata in C come:

```
int solveR(int *val, int *opt, int n, int k) {
    if (k==0)
        return 0;
    if (k==1)
        return val[1];
    return max(solveR(val,opt,n,k-1), solveR(val,opt,n,k-2) + val[k]);
}

void solve(int *val, int n) {
    int *opt;
    opt = calloc((n+1),sizeof(int));
    printf("Recursive solution: ");
    printf("maximum population covered %d\n", solveR(val, opt, n, n));
}
```

Questa soluzione porta alla seguente equazione alle ricorrenze:

$$T(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ T(n-1) + T(n-2) + 1 & n > 1 \end{cases}$$

identica a quella vista lezione per i numeri di Fibonacci, dunque la soluzione ricorsiva ha complessità esponenziale.

Passo 3: soluzione con programmazione dinamica bottom-up (calcolo del valore della soluzione ottima)

Ispirandosi alla formulazione ricorsiva della soluzione, la si trasforma in forma iterativa:

- $\text{opt}[0]$ e $\text{opt}[1]$ sono noti a priori,
- per $2 \leq i \leq n$ $\text{opt}[i] = \max(\text{opt}[i-1], \text{opt}[i-2] + \text{val}[i])$.

```
void solveDP(int *val, int n) {
    int i, *opt;
    opt = calloc((n+1),sizeof(int));
    opt[1] = val[1];
```



```
for (i=2; i<=n; i++) {  
    if (opt[i-1] > opt[i-2]+val[i])  
        opt[i] = opt[i-1];  
    else  
        opt[i] = opt[i-2] + val[i];  
}  
printf("Dynamic programming solution: ");  
printf("maximum population covered %d\n", opt[n]);  
displaySol(opt, val, n);  
}
```

Passo 4: costruzione della soluzione ottima

La funzione `displaySol` costruisce e visualizza la soluzione (città dove si installa un'antenna). Essa utilizza un vettore di interi `sol` di $n+1$ elementi per registrare se l'elemento i -esimo appartiene o meno alla soluzione. La decisione viene presa in base al contenuto del vettore `opt` e viene costruita mediante una scansione da destra verso sinistra, in verso quindi opposto alla scansione con cui `opt` è stato riempito dalla funzione `solveDP`. Il criterio per assegnare 0 o 1 alla cella corrente di `sol` rispecchia quello usato in fase di risoluzione:

- `sol[1]` è assunto valere 1, salvo modificare questa scelta nel corso dell'iterazione successiva
- se `opt[i]==opt[i-1]` è certo che nella città i -esima non è stata piazzata un'antenna, quindi `sol[i]=0`, mentre non si può dire nulla di `sol[i-1]`. L'iterazione quindi prosegue sulla città $(i-1)$ -esima
- se `opt[i] == opt[i-2] + val[i]` è certo che:
 - nella città i -esima è stata piazzata un'antenna, quindi `sol[i]=1`
 - nella città $(i-1)$ -esima non è stata piazzata un'antenna, quindi `sol[i-1]=0`avendo preso una decisione sia per la città i -esima che per la città $(i-1)$ -esima, l'iterazione prosegue sulla città $(i-2)$ -esima.

```
void displaySol(int *opt, int *val, int n){  
    int i, j, *sol;  
    sol = calloc((n+1), sizeof(int));  
    sol[1]=1;  
    i=n;  
    while (i>=2) {  
        printf("i=%d\n", i);  
        if (opt[i] == opt[i-1]){  
            sol[i] = 0;  
            i--;  
        }  
        else if (opt[i] == opt[i-2] + val[i]) {  
            sol[i] = 1;  
            sol[i-1] = 0;  
            i -=2;  
        }  
    }  
    for (i=1; i<=n; i++)  
        if (sol[i])  
            printf("%d ", val[i]);  
    printf("\n");
```



Esercizio n. 1: Sequenza di attività (versione 2)

Si consideri la situazione introdotta nell'esercizio n.1 del laboratorio 8. Si proponga una soluzione al medesimo problema sfruttando il paradigma della programmazione dinamica.

Suggerimento:

- si ordinino le attività lungo una linea temporale. Si definisca il criterio di ordinamento in base al vincolo del problema (gli intervalli della soluzione non devono intersecarsi)
- ispirandosi alla soluzione con programmazione dinamica del problema della Longest Increasing Sequence, si costruiscano soluzioni parziali considerando solamente le attività fino all' i -esima nell'ordinamento di cui sopra, definendo opportunamente secondo quale criterio considerare le attività. L'estensione di soluzioni parziali con l'introduzione di una attività aggiuntiva può essere fatta sulla base della compatibilità tra la "nuova" i -esima attività e le soluzioni già note ai problemi con le $i-1$ considerate precedentemente
- si seguano i passi svolti nell'esercizio precedente (dimostrazione di applicabilità, calcolo ricorsivo del valore ottimo, calcolo con programmazione dinamica bottom-up del valore ottimo e della soluzione ottima).

Esercizio n. 2: Gioco di ruolo (multi-file, con ADT)

[*Esercizio guidato: si forniscono architettura dei moduli ed esempi di file .h*]

A partire dal codice prodotto per l'esercizio n. 3 del laboratorio 8, lo si adatti così che sia il modulo personaggi sia il modulo inventario risultino ADT

La specifica "nessuna statistica può risultare minore di 1, indipendentemente dagli eventuali *malus* cumulativi dovuti alla scelta di equipaggiamento" può essere interpretata nelle seguenti 2 modalità, entrambe accettate:

- si impedisce la scelta di un equipaggiamento se il malus porta almeno una delle statistiche sotto il valore 1
- si ammette la scelta di malus che portano almeno una statistica a valori negativi o nulli, ma in questo caso questa statistica viene "mascherata" in fase di stampa, dove si stampa il valore fittizio 1, quando il valore è negativo o nullo.

Si tenga conto che esistono:

- un tipo di dato per un oggetto e un tipo di dato per un vettore di oggetti (inventario)
- un tipo di dato per un personaggio e un tipo di dato per una lista di personaggi
- un tipo di dato per un vettore di (riferimenti a) oggetti (equipaggiamento)

Si chiede di realizzare tutti i tipi di dato come ADT, utilizzando:

- la versione "quasi ADT" (struct visibile) per i tipi personaggio (pg_t) e oggetto (inv_t)
- la versione "ADT di I classe" per le collezioni, cioè il vettore di oggetti (invArray_t), la lista di personaggi (pgList_t) e gli equipaggiamenti (equipArray_t).

Per i riferimenti ad oggetti si evitino i puntatori, in quanto richiederebbero accesso a una struttura dati interna (all'ADT vettore di oggetti). Si utilizzino invece gli indici: ad un dato oggetto si fa



quindi riferimento mediante un intero (progressivo a partire da 0) che ne identifica la posizione nel vettore dell'inventario.

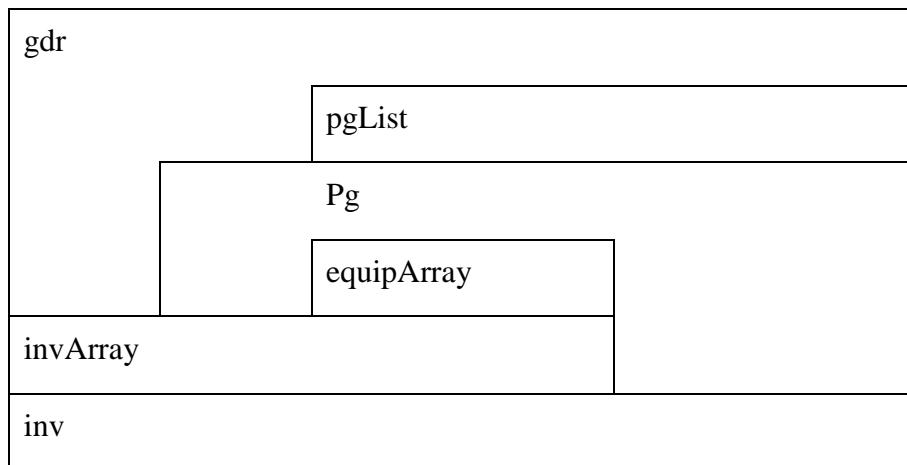
Pur potendo unificare l'ADT oggetto e il vettore inventario in un unico modulo, come pure l'ADT personaggio e l'ADT lista di personaggi, si consiglia di realizzare un modulo per ognuno degli ADT. Si realizzeranno quindi 5 file .c (pg.c, inv.c, pgList.c, invArray.c, equipArray.c) e 5 corrispettivi file .h per i moduli, più un .c per il client (ad esempio gdr.c).

I moduli pg ed inv realizzano tipi di dato composti per valore (sono quindi simili al tipo/ADT Item spesso usato in altri problemi): saranno ovviamente di dimensione ridotta.

I moduli per collezioni di dati dovranno fornire operazioni di creazione/distruzione, più eventuali operazioni di input o output, ricerca, modifica e/o cancellazione. Eventuali altre operazioni possono essere fornite qualora ritenute necessarie.

Architettura proposta:

La soluzione all'esercizio va realizzata secondo l'architettura dei moduli rappresentata nella seguente figura:



Lo schema mostra (dal basso in alto):

- gli elementi dell'inventario (modulo inv), un item quasi ADT, tipo inv_t, composto da 3 campi (di cui uno, un secondo tipo struct, stat_t, anch'esso quasi ADT) contiene le statistiche da leggere e aggiornare. Si consiglia di vedere inv.c come esempio di quasi ADT Item (non vuoto!) contenente operazioni elementari sui tipi di dato gestiti
- il vettore di elementi (invArray), realizzato come ADT vettore dinamico di elementi inv_t.
- il vettore di riferimenti (mediante indici) a elementi dell'inventario (equipArray) viene realizzato come semplice struct dinamica, contenente un vettore di interi di dimensione fissa (non allocato dinamicamente). Il modulo dipende da (è client di) invArray, solamente in quanto i riferimenti mediante indici sono relativi a un ADT di tale modulo
- il quasi ADT (un item) pg_t (modulo pg). Il modulo è client di inv, invArray e equipArray. Si noti che il quasi ADT contiene un campo ADT (equip) riferimento a un equipArray_t. Si tratta quindi di un quasi ADT di tipo 4 (in tal senso rappresenta un'eccezione rispetto alla prassi consigliata, non evitabile in base alle specifiche



dell'esercizio), avente cioè un campo soggetto ad allocazione e deallocazione. L'allocazione (`equipArray_init`) viene gestita nella funzione di lettura da file (`pg_read`), mentre per la deallocazione si è predisposta la funzione `pg_clean`, che chiama semplicemente la `equipArray_free` (il tipo `pg_t` non va deallocato, in quanto quasi ADT)

- l'ADT `pgList_t` (modulo `pgList`) è un ulteriore ADT di prima classe, che realizza una lista di elementi del modulo `pg`
- il modulo principale (`gdr`) è client di `pgList`, `pg` e di `invArray`.

Si allegano i `.h` dei vari moduli e il `.c` di `gdr`. A scelta, si possono eventualmente modificare i nomi di tipi e funzioni, nonché definizioni di tipi e parametri delle funzioni (pur di rispettare l'architettura proposta e le richieste).

Questo non è l'unico schema realizzabile, ma deriva dall'aver effettuato certe scelte di modularità e di ripartizione delle operazioni tra moduli. Si consiglia di esaminare le dipendenze tra i moduli, nonché le scelte fatte nell'assegnare operazioni e funzioni ai singoli moduli.

Si noti che le funzioni che gestiscono i quasi ADT in alcuni casi ricevono e/o ritornano `struct`, in altri casi riferimenti (puntatori) a `struct` (ad esempio le funzioni di input/output da file sono state spesso predisposte in modo da ricevere puntatori alla `struct` che riceve o da cui si prendono i dati coinvolti nell'IO. Sono possibili altre scelte (es. le `struct` passate sempre per valore).

ATTENZIONE

Si ricorda che un ADT di I classe NASCONDE i dettagli interni di un dato: NON E' QUINDI POSSIBILE A UN CLIENT ACCEDERE A TALI DETTAGLI: non sarà possibile, quindi la modifica dell'equipaggiamento di un dato personaggio da parte di un client in modo diretto, cioè ottenendo il puntatore all'elemento in lista, per accedere ai campi da modificare. Il client dovrà quindi, ad esempio, chiamare una opportuna funzione (fornita dal modulo `pgList`) avente come parametri il codice di un personaggio. Tale funzione, all'interno, effettuerà una ricerca dell'atleta e ne modificherà l'esercizio selezionato NON in modo diretto, ma chiamando, ad esempio, una funzione (fornita dal modulo `equipArray`), che riceve come parametri il nome dell'oggetto di interesse e il tipo di operazione da eseguire. Quest'ultima funzione cerca l'oggetto e modifica lo stato della struttura dati di conseguenza.

Valutazione: entrambi gli esercizi 1 e 2

Scadenza: caricamento di quanto valutato: entro le 23:59 del 20/12/2019.



Esercitazione di laboratorio n. 10

Esercizio n. 1: Collane e pietre preziose (versione 3)

Si risolva l'esercizio n. 1 del laboratorio 7 mediante il paradigma della memoization. Si richiede soltanto di calcolare la lunghezza massima della collana compatibile con le gemme a disposizione e con le regole di composizione.

Suggerimento: affrontare il problema con la tecnica del divide et impera, osservando che una collana di lunghezza P può essere definita ricorsivamente come:

- la collana vuota, formata da $P=0$ gemme
- una gemma seguita da una collana di $P-1$ gemme.

Poiché vi sono 4 tipi di gemme Z, R, T, S, si scrivano 4 funzioni f_Z , f_R , f_T , f_S che calcolino la lunghezza della collana più lunga iniziante rispettivamente con uno zaffiro, un rubino, un topazio e uno smeraldo avendo a disposizione z zaffiri, r rubini, t topazi e s smeraldi. Note le regole di composizione delle collane, è possibile esprimere ricorsivamente il valore di una certa funzione f_X sulla base dei valori delle altre funzioni.

Si presta attenzione a definire adeguatamente i casi terminali di tali funzioni, onde evitare di ricorrere in porzioni non ammissibili dello spazio degli stati.

Si ricorda che il paradigma della memoization prevede di memorizzare le soluzioni dei sottoproblemi già risolti in opportune strutture dati da progettare e dimensionare e di riusare dette soluzioni qualora si incontrino di nuovo gli stessi sottoproblemi, limitando l'uso della ricorsione alla soluzione dei sottoproblemi non ancora risolti.

Esercizio n. 2: Corpo libero

Il corpo libero è una specialità della ginnastica artistica/acrobatica in cui l'atleta deve eseguire una sequenza di elementi senza l'ausilio di attrezzi, ad eccezione della pedana di gara.

In un programma di corpo libero il ginnasta è tenuto ad eseguire una serie di passaggi acrobatici, detti diagonali.

Ogni diagonale è composta da uno o più elementi.

Ogni elemento è descritto da una serie di parametri:

- nome dell'elemento (una stringa di massimo 100 caratteri senza spazi)
- tipologia: l'elemento può essere un elemento acrobatico avanti [2], acrobatico indietro [1] o di transizione [0]
- direzione di ingresso: il ginnasta può entrare in un elemento frontalmente [1] o di spalle [0]
- direzione di uscita: il ginnasta può uscire da un elemento frontalmente [1] o di spalle [0]
- requisito di precedenza: l'elemento può essere eseguito come primo di una sequenza [0] o deve essere preceduto da almeno un altro elemento [1]
- finale: l'elemento non può essere seguito da altri elementi [1] o meno [0]
- valore: il punteggio ottenuto dall'atleta per la corretta esecuzione di un elemento (reale)
- difficoltà: la difficoltà di esecuzione dell'elemento (intero).

Gli elementi sono memorizzati in un file di testo (`elementi.txt`) in ragione di uno per riga. Il numero di elementi è riportato sulla prima riga del file.



Perché due elementi possano essere eseguiti in sequenza, la direzione di uscita del primo elemento deve coincidere con la direzione di ingresso del secondo elemento. Un ginnasta inizia una diagonale sempre frontalmente. La difficoltà di una diagonale è definita come la somma delle difficoltà degli elementi che la compongono. La difficoltà del programma di gara è data dalla somma delle difficoltà delle diagonali che lo compongono.

Ai fini dell'esercizio, si considerino le seguenti regole:

- il ginnasta deve presentare 3 diagonali
- il ginnasta deve includere almeno un elemento acrobatico in ogni diagonale
- il ginnasta deve includere almeno un elemento acrobatico avanti e almeno un elemento acrobatico indietro nel corso del suo programma, ma non necessariamente nella stessa diagonale
- il ginnasta deve presentare almeno una diagonale in cui compaiono almeno due elementi acrobatici in sequenza
- se il ginnasta include un elemento finale di difficoltà 8 o superiore nell'ultima diagonale presentata in gara, il punteggio complessivo della diagonale viene moltiplicato per 1.5
- ogni diagonale può contenere al massimo 5 elementi
- ogni diagonale non può avere difficoltà superiore a un dato valore DD
- il programma complessivamente non può avere difficoltà superiore a un dato valore DP.

Si scriva un programma in C in grado di identificare la sequenza di diagonali che permettono al ginnasta di ottenere il punteggio più alto possibile, dato un set di elementi disponibili e il valore dei due parametri DD e DP.

Esercizio n. 3: Corpo libero (vers. greedy)

A partire dallo scenario introdotto nell'esercizio precedente, si risolva il problema proponendo uno o più algoritmi greedy, definendo opportune funzioni obiettivo.

Esercizio n. 4: Rete di elaboratori

Un grafo non orientato e pesato rappresenta una rete di elaboratori appartenenti ciascuno ad una sottorete. Il peso associato ad ogni arco rappresenta il flusso di dati tra due elaboratori della stessa sottorete o di sottoreti diverse, come nell'esempio seguente (cfr. figura successiva).

Il grafo è contenuto in un file, il cui nome è passato come argomento sulla linea di comando. Il file è composto da un numero indefinito di righe ciascuna delle quali contiene una quaterna di stringhe alfanumeriche, di al massimo 30 caratteri, e un intero:

```
<id_elab1> <id_rete1> <id_elab2> <id_rete2> <flusso>
```

Si facciano anche le seguenti assunzioni:

- i nomi dei singoli nodi sono univoci all'interno del grafo
- non sono ammessi cappi
- tra due nodi c'è al massimo un arco (non è un multigrafo)
- le sotto-reti sono sotto-grafi non necessariamente connessi.



Si scriva un programma in C in grado di caricare in memoria il grafo, leggendone i contenuti da file e di potervi effettuare alcune semplici operazioni.

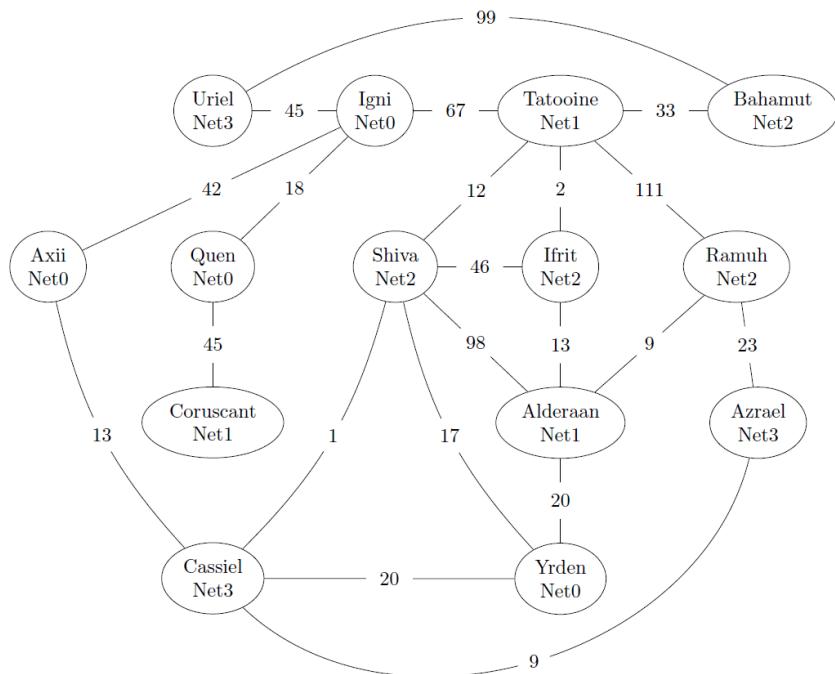
La rappresentazione della struttura dati in memoria deve essere fatta tenendo conto dei seguenti vincoli:

- il grafo sia implementato come ADT di I classe, predisposto in modo tale da poter contenere sia la matrice sia le liste di adiacenza. Nella fase di caricamento dei dati da file si generi solamente la matrice di adiacenza, su comando esplicito va generata anche la lista di adiacenza
- si utilizzi una tabella di simboli tale da fornire corrispondenze “*da nome a indice*” e “*da indice a nome*”.

Sul grafo, una volta acquisito da file, sia possibile:

- elencare in ordine alfabetico i vertici e per ogni vertice gli archi che su di esso insistono, sempre in ordine alfabetico
- dati 3 vertici i cui nomi sono letti da tastiera, verificare se essi sono adiacenti a coppie, cioè se formano un sottografo completo. Tale funzione sia implementata sia per la rappresentazione con matrice delle adiacenze, sia per la rappresentazione con lista delle adiacenze
- generare la rappresentazione a lista di adiacenza, **SENZA** leggere nuovamente il file, a partire da quella a matrice di adiacenza.

In allegato al testo è presente il grafo d'esempio (`grafo.txt`) rappresentato a seguire:



Valutazione: gli esercizi 1, 3 e 4 saranno oggetto di valutazione

Scadenza: caricamento di quanto valutato: entro le 23:59 del 24/01/2020



Esercitazione di laboratorio n. 11

Esercizio n.1: Titoli azionari

Versione estesa del compito d'esame del 13/09/2018

I titoli azionari delle aziende quotate in Borsa possono essere oggetto di transazioni (vendita/acquisto di un certo numero di titoli in una certa data/ora ad un certo valore su una certa piazza) in una o più piazze finanziarie in tutto il mondo. Si immagini di voler creare un sistema globale, cioè che tenga conto di tutte le Borse, per la memorizzazione e gestione dei titoli azionari.

Ogni Borsa invia al sistema con regolarità un file con un elenco di transazioni raggruppate per titolo. Sulla prima riga compare il numero complessivo di titoli (nel file), di seguito i titoli e le relative transazioni con il seguente formato (ripetuto per ognuno dei titoli):

- una prima riga contiene il <titolo> (codice alfanumerico univoco di al più 20 caratteri) seguito dal numero di transazioni
- sulle righe successive, una per riga, le transazioni relative al titolo, sotto forma di quaterne <data> <ora> <valore> <numero>. Le date sono nella forma aaaa/mm/gg, le ore sono nel formato su 24 ore hh:mm riferite al tempo di Greenwich (GMT), mentre i valori dei titoli sono rappresentati con numeri reali non negativi, la quantità è un intero.
- non si presupponga nessuna forma di ordinamento né sui titoli, né sulle transazioni.

Il sistema acquisisce i file e ne memorizza i contenuti in un'opportuna struttura dati a 2 livelli che prevede (primo livello) una collezione di titoli e per ogni titolo (secondo livello) una collezione delle sue quotazioni giornaliere. La quotazione giornaliera Q_i di un titolo in una certa data i è la media di tutti i valori v_{ij} di quel titolo in quella data pesati sul numero di titoli scambiati n_{ij}

$$Q_i = \frac{\sum_j v_{ij} \cdot n_{ij}}{\sum_j n_{ij}}$$

Si noti che, identificando con j la j -esima transazione del titolo alla data i , v_{ij} è il valore del titolo nella transazione, mentre n_{ij} è la quantità di titoli scambiati in tale transazione.

Si realizzino:

- un quasi ADT per la data e uno per l'ora (o un unico ADT per entrambe), implementati come struct con campi interi per anno, mese e giorno, ore e minuti
- un ADT di I classe per il titolo e uno per collezione di titoli (a scelta se in un solo modulo o in due). Per la collezione di titoli si faccia uso di una lista ordinata (si usi il codice del titolo come chiave di ordinamento)
- un quasi ADT per la quotazione giornaliera e un ADT di I classe per collezione di quotazioni giornaliere (a scelta se in un solo modulo o in due). Per la collezione di quotazioni si faccia uso di un BST (con data come chiave di ricerca e ordinamento). Per gli inserimenti di dati nel BST sono sufficienti gli inserimenti in foglia
- un client che fornisca le seguenti funzionalità:
 1. acquisizione del contenuto di un file contenente un insieme di transazioni
 2. ricerca di un titolo azionario (ricerca in lista)
 3. ricerca, dato un titolo precedentemente selezionato, della sua quotazione in una certa data (ricerca in un BST)
 4. ricerca, dato un titolo precedentemente selezionato, della sua quotazione minima e massima in un certo intervallo di date (si noti che la ricerca, in un BST, di più chiavi comprese in un dato intervallo, non è una funzione standard e va quindi realizzata: si consiglia una variante di un algoritmo di visita in-order)



5. ricerca, dato un titolo precedentemente selezionato, della quotazione minima e massima lungo tutto il periodo registrato (il problema può essere ricondotto a un caso particolare del punto precedente)
6. dato un titolo precedentemente selezionato, bilanciamento dell'albero di quotazioni se il rapporto tra il cammino più lungo e più corto nell'albero supera una certa soglia S .

Nota: si ricorda che è possibile bilanciare un BST dato mediante applicazione ricorsiva del partizionamento rispetto alla chiave mediana. (*Per la soluzione di questo punto è necessario il materiale completo sui BST, che verrà presentato nelle lezioni della settimana 7-10/1/2020*)

Valutazione: l'esercizio sarà oggetto di valutazione

Scadenza: caricamento di quanto valutato: entro le 23:59 del 24/01/2020.



Esercitazione di laboratorio n. 12

Esercizio n.1: Grafi e DAG

Un grafo pesato, orientato e connesso è memorizzato in un file di testo `grafo.txt` con il seguente formato:

- sulla prima riga un intero N rappresenta il numero di vertici del grafo
- seguono N righe ciascuna delle quali contiene una stringa alfanumerica, di al massimo 30 caratteri rappresentante l'identificatore univoco del nodo
- seguono un numero indefinito di terne $\langle id_1 \rangle \langle id_2 \rangle \langle peso_arco \rangle$ a rappresentare gli archi orientati del grafo. Il peso è un valore intero non negativo.

Si scriva un programma C che svolga le seguenti operazioni:

- individuazione di tutti gli insiemi di archi di cardinalità minima la cui rimozione renda il grafo originale un DAG
- costruzione di un DAG rimuovendo, tra tutti gli insiemi di archi generati al passo precedente, quelli dell'insieme a peso massimo. Il peso di un insieme di archi è definito come la somma dei pesi degli archi in esso contenuti
- calcolo delle distanze massime da ogni nodo sorgente verso ogni nodo del DAG costruito al passo precedente (cfr lucidi cap. 19).

Valutazione: l'esercizio sarà oggetto di valutazione

Scadenza: caricamento di quanto valutato: entro le 23:59 del 24/01/2020.