Allocazione dinamica

1.1

Dato in ingresso un array di interi dall'utente (dovrà essere allocato dinamicamente, chiedendo la lunghezza all'utente), scrivere una funzione `filter` ricorsiva, la quale ritorna un array contenente solo gli elementi pari contenuti nell'array in input.

Firma delle funzioni da scrivere

- `int* filter(int*, int, int&)`
- `int* filter_rec(int*, int, int, int&)`

1.2

Dato in ingresso le dimensioni di una matrice di interi, scrivere una funzione che restituisca la matrice trasposta.

Note

- Ricordarsi di **deallocare**!
- La matrice deve essere inizializzata con valori random tra `[0,10]`
- La funzione che calcola la trasposta **non** deve essere ricorsiva

1.3

Scrivere un programma che chieda all'utente le dimensioni di una matrice di interi, e prenda i valori sempre da terminale. Una volta inserita la matrice a schermo deve essere mostrato l'elemento dal valore massimo per ciascuna riga.

```
Number of rows: 3
Number of cols: 3
Insert values:
1 2 3 4 5 6 7 8 9
1 2 3
4 5 6
7 8 9
max row 1 => 3
max row 2 => 6
max row 3 => 9
```

Strutture dati

2.1

Scrivere un programma che gestisca la collisione in una dimensione tra due blocchi caratterizzati da due proprietà:

- massa
- velocità

La formula per calcolare la velocità finale è la seguente:

$$V_1 = -\frac{m_2 \cdot v_2}{m_1}$$
$$V_2 = -\frac{m_1 \cdot v_1}{m_2}$$

Dove m_1 e m_2 sono le masse dei due blocchi, v_1 e v_2 le velocità iniziali e V_1 e V_2 le velocità finali.

Basta che il programma gestisca una sola collisione e stampi le velocità finali.

2.2

Un'agenzia del farmaco ha un grosso database per categorizzare i farmaci che usa. Il database memorizza le seguenti informazioni: id del farmaco, numero di molecole, numero di altri farmaci con cui interagisce, numero di test condotti, numero di reazioni avverse.

Il database può essere scaricato dal Google Drive.

```
id numero_molecole numero_interazioni numero_test numero_reazioni
```

Leggere il file, caricare i dati in memoria e successivamente dare le seguenti informazioni:

- Numero totale di farmaci;
- Qual è il farmaco con il maggior numero di reazioni avverse;
- Qual è il farmaco più pericoloso, ossia quello con il rapporto (reazioni avverse)/(test condotti) più alto;
- Quale farmaco contiene il maggior numero di molecole;
- Quali sono i farmaci che hanno un numero di interazioni con altri farmaci sopra la media.

2.3 – Facile

Scrivere un programma per permettere la ricerca in un albero genealogico tramite il nome di una persona. Definire una struct `Persona` con gli attributi `nome`,`madre`, e `padre`. Memorizzare le struct delle persone in un array `Persone`. Se una persona non ha madre o padre, lasciare il campo vuoto.

Inizializzare un albero genealocico con dei valori randomici.

Stampare a video qual è il genitore con più figli. Per fare questo potete aggiornare i campi della struttura, aggiungerne o toglierne.

Cercare poi di trovare un algoritmo che permetta di scoprire anche chi è il nonno/a con più nipoti senza però modificare i campi della struttura ulteriormente.

Per un esempio più completo, potete scaricare il database "albero_genealogico.txt" dal Google Drive dove la prima riga indica il numero di Persone nel dataset e quelle seguenti sono nella forma `nome genitore1 genitore2`.

2.3 – Difficile

Scrivere un programma per permettere la ricerca in un albero genealogico tramite il nome di una persona. Definire una struct `Persona` con gli attributi `nome`,`madre`, `padre`, dove `madre` e `padre` sono puntatori a struct di tipo `Persona`.
Scrivere una funzione `cercaPersona (Persona**, const char*)` che prenda in input un puntatore all'albero e una stringa e cerchi se esiste una persona con quel nome (usare una procedura ricorsiva).

Aiuto:

- inizializzare le persone che chiudono l'albero (es. i nonni) con `NULL`, in modo da poter controllarne la presenza ed evitare seg. fault.
- può essere utile usare `typedef` per semplificare l'uso dei puntatori, ad esempio

```
1 struct Persona {...};
2 typedef Persona * PuntatorePersona;          // punta alla singola
3                                              // istanza di Persona
4 typedef PuntatorePersona * AlberoPersona;    // punta all'intero
5                                              // albero di Persona
```

in questo caso la funzione avrà una firma del tipo `cercaPersona (AlberoPersona, const char*)`.

2.4

Scrivere un programma che permetta la gestione di una serie di macchine tramite `LinkedList`, definita nel seguente modo:

```
struct LinkedList {
 // attributi utili
 // (...)
 // puntatore all'item successivo nella lista
 LinkedList * next;
};
```

2.4.1

Scrivere una funzione che permetta di stampare tutti gli item all'interno della lista.

2.4.2

Scrivere una funzione che ricerchi un determinato item all'interno della lista e lo rimuova.

N.B: non è necessario de-allocare l'oggetto, basta manipolare il campo `next`

Usare la funzione creata in `4.1` per controllare che l'oggetto sia effettivamente stato rimosso

2.5

Creare un programma per gestire una macchinetta del caffè a Povo, che contenga i campi `caffe` e `credito` che indichino rispettivamente il numero di caffè rimanenti e il credito dell'utente. Prendere in input i valori iniziali.

Implementare le funzioni

- `void addCoin(CoffeeMachine * machine, int val)` che aggiunge `val` centesimi al credito
- `bool getCoffee(CoffeeMachine * machine)` che restituisce `true` se è possibile erogare un caffè (ovvero se il credito è almeno 39 centesimi e ci sono caffè rimanenti), altrimenti restituisce `false` e non eroga il caffè.

Creare poi un while loop che continui a chiedere all'utente se vuole inserire monete, se vuole prendere un caffè o se vuole sapere il saldo.

Estendere poi il programma a gestire `N` macchinette del caffè con `N` preso in input all'inizio del programma. Aggiungere i dovuti controlli.