Esame di Laboratorio di Fondamenti di Informatica II e Lab. del 26/06/2020

Note importanti:

- È considerato errore qualsiasi output non richiesto dagli esercizi.
- È consentito utilizzare funzioni ausiliarie per risolvere gli esercizi.
- Quando caricate il codice sul sistema assicuratevi che siano presenti tutte le direttive di include necessarie, comprese quelle per l'utilizzo delle primitive. Non dovete caricare l'implementazione delle primitive.
- È importante sviluppare il codice in Visual Studio, prima del caricamento sul sistema, per poter effettuare il debug delle funzioni realizzate!

Esercizio 1 (8 punti)

In matematica, una *frazione egizia* o *frazione egiziana* è una frazione scritta come somma di frazioni unitarie, ovvero frazioni aventi numeratore unitario e denominatore intero positivo.

Una *frazione egizia* è quindi una frazione del tipo:

$$\frac{1}{a_1} + \frac{1}{a_2} + ... + \frac{1}{a_n}$$

con $a_1, a_2, ..., a_n$, interi positivi a due a due distinti. Questa notazione veniva usata dagli antichi *egizi* da cui prende appunto il nome.

Qualunque frazione N/D con $D > N$ può essere scritta sotto forma di *frazione egizia* utilizzando il seguente algoritmo ricorsivo:

1. si calcola la più grande frazione unitaria secondo la formula $\frac{1}{\lceil \frac{D}{N} \rceil}$;
2. se D è un multiplo di N l'algoritmo termina, altrimenti si procede ricorsivamente per $\frac{N'}{D'} = \frac{N}{D} - \frac{1}{\lceil \frac{D}{N} \rceil}$.

Data ad esempio la frazione $7/24$, l'algoritmo procede come segue:

1. calcola la prima frazione unitaria, ovvero $\frac{1}{\lceil \frac{24}{7} \rceil} = 1/4$;
2. procede ricorsivamente per $7/24 - 1/4 = 4/96$;
3. calcola la seconda frazione unitaria, ovvero $\frac{1}{\lceil \frac{96}{4} \rceil} = 1/24$;
4. l'algoritmo termina in quanto 96 è multiplo di 4.

quindi la conversione in *frazione egizia* risulta essere $\frac{1}{4} + \frac{1}{24}$

Nel file `egizia.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern void FrazioneEgizia(int n, int d)
```

La procedura prende in input due interi `n` e `d` che rappresentano rispettivamente numeratore e denominatore della frazione da convertire in *frazione egizia*. La funzione deve convertire la frazione data in *frazione egizia* e stamparla su standard output secondo il formato:

```
1/a1 + 1/a2 + ... + 1/an
```

Quindi con `n = 7` e `d = 24` dell'esempio precedente la funzione deve stampare `1/4 + 1/24`

Se $n \leq 0$, oppure $d \leq 0$, oppure $n \geq d$ la funzione deve stampare su standard output la stringa `Impossibile convertire la frazione data in frazione egizia.`

Non saranno considerate valide soluzioni che non fanno uso della ricorsione per la generazione della *frazione egizia*.

Esercizio 2 (10 punti)

Nel file `cattura.c` definire la procedura corrispondente alla seguente dichiarazione:

```
extern int* CacciatorePreda(const char *v, size_t v_size, int u);
```

La funzione prende in input un vettore di caratteri, `v`, la sua dimensione `v_size`, e una distanza `u`. Ogni elemento del vettore contiene un cacciatore, identificato dal carattere `'C'` (maiuscolo o minuscolo) o una preda, identificata dal carattere `'P'` (maiuscolo o minuscolo). La distanza tra un cacciatore e una preda corrisponde alle loro distanze nel vettore. Quindi un cacciatore in posizione `1` dista `2` unità da una preda in posizione `3`.

Un cacciatore può catturare una preda solo se questa si trova ad una distanza uguale o inferiore di `u` unità da lui. Inoltre, ogni cacciatore può catturare al massimo una preda.

Utilizzando un algoritmo di backtracking, la funzione `CacciatorePreda` deve associare ad ogni cacciatore una preda, in modo tale che sia massimo il numero di prede catturate. La funzione deve quindi ritornare un vettore di interi, allocato dinamicamente, della stessa dimensione di `v`. Ogni elemento del vettore deve contenere `-2` se l'elemento corrisponde ad una preda catturata, `-1` se l'elemento corrisponde ad una preda libera o a un cacciatore senza preda, l'indice della preda catturata se l'elemento corrisponde ad un cacciatore che ha catturato una preda.

Dato ad esempio il vettore `v` che segue e `u = 1` la funzione dovrebbe ritornare il vettore `s` o un vettore equivalente:

```
       0   1   2   3   4
v = {'C', 'P', 'P', 'C', 'P'}
s = { 1 , -2 , -2 ,  2 , -1 }
```

In questo caso il cacciatore in posizione `0` ha catturato la preda in posizione `1`, il cacciatore in posizione `3` ha catturata la preda in `2`. Ovviamente esistono soluzioni equivalentemente ottime, che sarebbero considerate valide, ad esempio `s' = {1, -2, -1, 4, -2 }`.

Si consiglia l'uso di una funzione ausiliaria per la risoluzione di questo esercizio.

Esercizio 3 (6 punti)

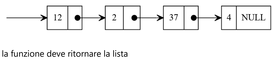
Nel file `minori.c` definire la procedura corrispondente alla seguente dichiarazione:

```
extern Item* CreaDaMinori(const Item* i, int v);
```

La funzione prende in input una lista di numeri interi (puntatore alla testa) e un valore intero `v`. La funzione deve **creare una nuova lista** contenente tutti e soli gli elementi di quella di input di valore minore o uguale a `v`. La funzione deve quindi ritornare la nuova lista (puntatore alla testa). L'ordine degli elementi deve essere preservato.

Se la lista di input è vuota o non contiene elementi che rispettano i vincoli, la funzione deve ritornare `NULL`.

Ad esempio, dato `v = 5` e la lista



la funzione deve ritornare la lista

Per la risoluzione di questo esercizio avete a disposizione le seguenti definizioni:

```
typedef int ElemType;

struct Item{
    ElemType value;
    struct Item *next;
};
typedef struct Item Item;
```

e le seguenti funzioni primitive e non:

```
int ElemCompare(const ElemType *e1, const ElemType *e2);
ElemType ElemCopy(const ElemType *e);
void ElemDelete(ElemType *e);
int ReadElem(FILE *f, ElemType *e);
int ReadStdinElem(ElemType *e);
void WriteElem(const ElemType *e, FILE *f);
void WriteStdoutElem(const ElemType *e);


Item* CreateEmptyList(void);
Item* InsertHeadList(const ElemType *e, Item* i);
bool IsEmptyList(const Item *i);
const ElemType* GetHeadValueList(const Item *i);
Item* GetTailList(const Item* i);
Item* InsertBackList(Item* i, const ElemType *e);
void DeleteList(Item* item);
void WriteList(const Item *i, FILE *f);
void WriteStdoutList(const Item *i);
```

Trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file `list_int.h` e `list_int.c` scaricabili da OLJ, così come la loro documentazione.

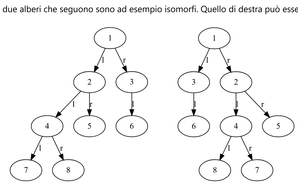
Esercizio 4 (9 punti)

Nel file `isomorfi.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern bool Isomorfi(const Node *t1, const Node *t2);
```

La funzione prende in input due alberi binari di interi `t1` e `t2` (puntatori ai nodi radice) e deve ritornare `true` se i due alberi sono isomorfi, `false` altrimenti. Due alberi binari sono isomorfi se uno dei due può essere ottenuto a partire dall'altro scambiando il figlio destro con il figlio sinistro di un numero arbitrario di nodi. Due alberi vuoti sono isomorfi.

I due alberi che seguono sono ad esempio isomorfi. Quello di destra può essere infatti generato da quello di sinistra scambiando i sottoalberi del nodo `1` e i sottoalberi del nodo `4`.



N.B. La funzione non deve trasformare gli alberi, ma verificare se questi sono o meno isomorfi.

Per la risoluzione di questo esercizio avete a disposizione le seguenti definizioni:

```
typedef int ElemType;

struct Node{
    ElemType value;
    struct Node *left;
    struct Node *right;
};
typedef struct Node Node;
```

e le seguenti funzioni primitive e non:

```
int ElemCompare(const ElemType *e1, const ElemType *e2);
ElemType ElemCopy(const ElemType *e);
void ElemDelete(ElemType *e);
int ReadElem(FILE *f, ElemType *e);
int ReadStdinElem(ElemType *e);
void WriteElem(const ElemType *e, FILE *f);
void WriteStdoutElem(const ElemType *e);

Node* CreateEmptyTree(void);
Node* CreateRootTree(const ElemType *e, Node* l, Node* r);
bool IsEmptyTree(const Node *n);
const ElemType* GetRootValueTree(const Node *n);
Node* LeftTree(const Node *n);
Node* RightTree(const Node *n);
bool IsLeafTree(const Node *n);
void DeleteTree(Node *n);

void WritePreOrderTree(const Node *n, FILE *f);
void WriteStdoutPreOrderTree(const Node *n);
void WriteInOrderTree(const Node *n, FILE *f);
void WriteStdoutInOrderTree(const Node *n);
void WritePostOrderTree(const Node *n, FILE *f);
void WriteStdoutPostOrderTree(const Node *n);
```

Trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file `tree_int.h` e `tree_int.c` scaricabili da OLJ, così come la loro documentazione.