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)

La dichiarazione della funzione di libreria `atoi()` è la seguente:

```
int atoi(const char *str);
```

La funzione prende in input un stringa `C` zero terminata e ritorna la sua conversione a `int`.

In questo esercizio vi si chiede di implementare una versione **ricorsiva** semplificata della funzione `atoi()`. La funzione dovrà avere la seguente dichiarazione:

```
int AToI(const char *str);
```

Se la stringa è `NULL` la funzione deve ritornare `0`, in tutti gli altri casi deve essere ritornato un `int` contenente il valore intero corrispondente alla stringa. Si assuma che la stringa sia sempre formata da soli caratteri numerici.

Non saranno considerate valide soluzioni che non fanno uso della ricorsione per la conversione della stringa.

Esercizio 2 (10 punti)

Creare i file `colora.h` e `colora.c` che consentano di utilizzare la seguente struttura:

```
struct Matrix{
    size_t size;
    bool *data;
};
```

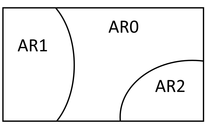
e la funzione:

```
extern int MappaColori(const struct Matrix *m, const char *c, size_t c_size)
```

La `struct` consente di rappresentare matrici quadrate di valori booleani, che rappresentano le adiacenze tra aree numerate da `0` a `size - 1` di una di mappa. La seguente matrice di adiacenze

	AR0	**AR1**	**AR2**
AR0	1	1	1
AR1	1	1	0
AR2	1	0	1

ci dice, ad esempio, che l'`Area0` è adiacente all'`Area1` e all'`Area2`, ma che `Area1` e `Area2` non sono adiacenti tra loro. Una possibile rappresentazione della mappa è la seguente:



La funzione prende in input una matrice di adiacenze, `m`, e un vettore di caratteri, `c`, che rappresenta un set di colori. La dimensione del vettore `c` e quindi il numero di colori che si hanno a disposizione è dato dal parametro `c_size`. Per semplicità ogni colore è rappresentato da un solo carattere: la lettera iniziale di quel colore.

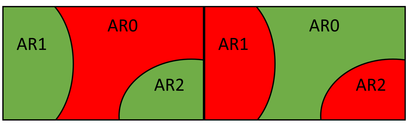
Utilizzando un algoritmo di backtracking la funzione `MappaColori` deve individuare e visualizzare su standard output tutti i modi in cui è possibile colorare una mappa di aree utilizzando solo i colori disponibili, e garantendo che due aree adiacenti non abbiamo mai lo stesso colore.

La funzione deve anche ritornare il numero di soluzioni trovate. Il formato dell'output dovrà coincidere con quello dell'esempio che segue.

Sia data la matrice di adiacenze sopra illustrata e siano dati i due colori rosso e verde: `c = {"r", "v"}`. La funzione deve produrre il seguente output:

```
0 -> r; 1 -> v; 2 -> v;
0 -> v; 1 -> r; 2 -> r;
```

La mappa dell'esempio può infatti essere colorata solo in due modi:



L'ordine con cui vengono stampate le soluzioni non conta. Si consiglia l'uso di una funzione ausiliaria per la risoluzione di questo esercizio.

Esercizio 3 (6 punti)

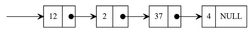
Nel file `copia.c` definire la procedura corrispondente alla seguente dichiarazione:

```
extern Item* CopiaDaN(const Item* i, int n);
```

La funzione prende in input una lista di numeri interi (puntatore alla testa) e un valore intero `n`. La funzione deve **creare una nuova lista** contenente tutti gli elementi di quella di input, a partire dall'`n`-esimo *incluso* (si parte a contare da 1). 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 contiene meno di `n` elementi, la funzione deve ritornare `NULL`.

Ad esempio, dato `n = 2` 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 `equivalenti.c` definire la funzione corrispondente alla seguente dichiarazione:

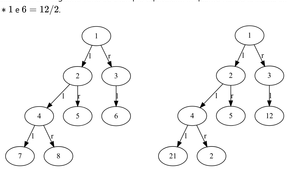
```
extern bool Equivalenti(const Node *t1, const Node *t2);
```

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 *equivalenti*, `false` altrimenti. Due alberi binari sono *equivalenti* se soddisfano le seguenti condizioni:

1. I due alberi hanno la stessa struttura;
2. I nodi non foglia contengono gli stessi valori.
3. I nodi foglia corrispondenti contengono valori uno multiplo dell'altro.

Due alberi vuoti sono equivalenti.

I due alberi che seguono sono ad esempio equivalenti in quanto hanno la stessa struttura e tutti i nodi, escluse le foglie, contengono gli stessi valori. Inoltre, per le foglie corrispondenti si ha: $7 = 21/3$, $8 = 2 * 4$, $5 = 5 * 1$ e $6 = 12/2$.



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.