Esame di Laboratorio del 29/06/2022

Note importanti:

- È considerato errore qualsiasi output non richiesto dagli esercizi.
- È consentito utilizzare funzioni ausiliarie per risolvere gli esercizi (in alcuni casi è caldamente consigliato o indispensabile!).
- 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 (o altri IDE) prima del caricamento sul sistema, così da poter effettuare il debug delle funzioni realizzate!
- Su OLJ non sarà possibile eseguire più di una compilazione/test ogni 3 minuti, per un massimo di 3 compilazioni per esercizio. Il numero di sottomissioni, invece, non è sottoposto a vincoli temporali o quantitativi.

Esercizio 1

Scrivere un programma a linea di comando con la seguente sintassi:

```
sin-x <x> <i>
```

Il programma deve calcolare **ricorsivamente** un'approssimazione del valore di $sin(x)$ utilizzando lo sviluppo in serie di Taylor-Mc Laurin fino al termine i-esimo:

$$sin(x) \approx \sum_{n=0}^{i} \frac{(-1)^n}{(2n+1)!} \cdot x^{2n+1} \qquad \forall x \in \mathbb{R}$$

Si noti che per $i \to \infty$, lo sviluppo in serie sopra riportato non rappresenterebbe un'approssimazione, ma bensì il reale valore di $sin(x)$.

Se il numero di parametri passati al programma è sbagliato o se `i` è minore di `0` il programma termina con codice `1`, in tutti gli altri casi termina con codice `0`. Prima di terminare, se possibile, il programma stampa su standard output l'approssimazione di $sin(x)$ ottenuta.

Si consiglia di utilizzare internamente dei `double` per tutti i calcoli, così da avere una precisione sufficiente a contenere il risultato.

Ad esempio, invocando il programma `six-x 1.570796326 4` su standard output deve essere stampato il valore `1.000004`

Non saranno considerate valide soluzioni che non fanno uso della ricorsione per il calcolo dell'approsimazione di $sin(x)$.

Esercizio 2

Nel file `parentesi.c` si realizzi la funzione:

```
extern int Parentesi(int n);
```

La funzione prende in input un numero interno, `n`. La funzione deve, utilizzando un algoritmo di *backtracking*, stampare su standard output tutti i modi con cui `n` coppie di parentesi possono essere combinate correttamente. Al termine, la funzione ritorna il numero di soluzioni trovate.

Ogni soluzione deve essere separata dalla precedente dal carattere <a capo> e il formato deve corrispondere a quello dell'esempio che segue.

L'ordine con cui vengono visualizzate le soluzioni non è significativo, ma eventuali soluzioni duplicate devono essere scartate.

Se `n` è negativo la funzione termina senza stampare nulla e ritorna `-1`.

Esempio:

```
n = 3
```

La funzione deve stampare, non necessariamente in questo ordine:

```
((()))
(()())
(())()
()(())
()()()
```

e ritornare 5.

Esercizio 3

Nei file `cc.h` e `cc.c` si implementi la definizione della seguente funzione:

```
extern int ComponentiConnesse(const Item *i, const ElemType* v, size_t v_size);
```

La funzione prende in input una lista di `ElemType`, `i`, contenente valori univoci, un vettore di `ElemType`, `v`, e la sua dimensione `v_size`. Il vettore `v` contiene un sottoinsieme dei valori contenuti in `i`.

La funzione calcola e ritorna il numero delle componenti connessere in `v`. Due valori sono connessi tra loro se nella lista compaiono consecutivamente.

Ad esempio, dati `i = [0, 1, 2, 3, 4]` e `v = {1, 2, 4}`, la funzione deve ritornare 2. Infatti, i valori 1 e 2 sono connessi in quanto consecutivi nella lista `i` e formano una componente. Il valore 4, non connesso ai precedenti, forma da solo la seconda componente.

Dati invece `i = [0, 1, 3, 4, 5, 6, 8, 7]` e `v = {7, 8, 0, 4, 6}`, la funzione deve ritornare 3. Infatti, le componenti connesse sono:

```
[0]
[4]
[6,8,7]
```

Sia la lista che il vettore di input potrebbero essere vuoti. In entrambi i casi le componenti connesso sono 0.

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:

```
ElemType ElemCopy(const ElemType *e);
void ElemSwap (ElemType *e1, ElemType *e2)
void ElemDelete(ElemType *e);
void ElemWrite(const ElemType *e, FILE *f);
void ElemWriteStdout(const ElemType *e);

Item *ListCreateEmpty(void);
Item *ListInsertHead(const ElemType *e, Item* i);
bool ListIsEmpty(const Item *i);
const ElemType *ListGetHeadValue(const Item *i);
Item *ListGetTail(const Item *i);
Item *ListInsertBack(Item *i, const ElemType *e);
void ListDelete(Item *item);
void ListWrite(const Item *i, FILE *f);
void ListWriteStdout(const Item *i);
```

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

Esercizio 4

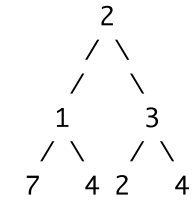
Nel file `percorso_somma.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern bool PercorsoSomma(Node *t, const ElemType *target);
```

La funzione prende in input un albero binario di `int`, `t`, e un `target`. La funzione ritorna `true` se esiste (almeno) un percorso dalla radice ad una foglia la somma delle cui chiavi è esattamente `target`. La funzione ritorna `false` altrimenti.

Se l'abero di input è vuoto la funzione ritorna `false`.

Dato ad esempio l'albero:



Se `target = 7` la funzione ritornerebbe `true` dato che il percorso 2->1->4 somma 7.

Se `target = 9` la funzione ritornerebbe `true` dato che nessuno dei 2->3->4 somma 9.

Se `target = 11` la funzione ritornerebbe `false` dato che nessuno dei percosi dalla radice alle foglie somma 11.

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 ElemRead(FILE *f, ElemType *e);
int ElemReadStdin(ElemType *e);
void ElemWrite(const ElemType *e, FILE *f);
void ElemWriteStdout(const ElemType *e);

Node *TreeCreateEmpty(void);
Node *TreeCreateRoot(const ElemType *e, Node *l, Node *r);
bool TreeIsEmpty(const Node *n);
const ElemType *TreeGetRootValue(const Node *n);
Node *TreeLeft(const Node *n);
Node *TreeRight(const Node *n);
bool TreeIsLeaf(const Node *n);
void TreeDelete(Node *n);

void TreeWritePreOrder(const Node *n, FILE *f);
void TreeWriteStdoutPreOrder(const Node *n);
void TreeWriteInOrder(const Node *n, FILE *f);
void TreeWriteStdoutInOrder(const Node *n);
void TreeWritePostOrder(const Node *n, FILE *f);
void TreeWriteStdoutPostOrder(const Node *n);
```

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

Esercizio 5

L'algoritmo di ordinamento *BrickSort* o *Even-OddSort* è un algoritmo di ordinamento originariamente sviluppato per l'uso su processori paralleli. L'algoritmo si basa sull'algoritmo bubble-sort. In sostanza è costituito da due iterazioni principali, la prima confronta tutte le coppie di posizione dispari (coppie il cui primo elemento è in posizione dispari). Se una coppia è nell'ordine sbagliato gli elementi vengono scambiati. L'iterazione successiva, invece, viene applicata alle coppie di posizione pari (coppie il cui primo elemento è in posizione pari). Queste due iterazioni vengono alternate e, come nel bubble sort, ripetute fino a quando il vettore non è ordinato.

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

```
extern void BrickSort(int* v, size_t v_size);
```

La funzione prende in input un vettore di `int`, `v`, e la sua dimenzione in numero di elementi, `v_size`. La funzione ordina il vettore `v` utilizzando l'algoritmo di ordinamento appena descritto.