Esame di Laboratorio del 13/02/2023

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

Il numero π può essere calcolato facendo uso di serie numeriche, come ad esempio la serie infinita:

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ ... \tag{1}$$

Una buona approssimazione di π si ottiene sommando n termini di questa serie, che può essere scritta in forma ricorsiva come:

$$\pi_0 = 4 \tag{2}$$

$$\pi_n = \frac{4 * (-1)^n}{2 * n + 1} + \pi_{n-1} \tag{3}$$

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

```
pigreco <n>
```

Il programma prende in input un numero intero positivo `n`, e, utilizzando la funzione ricorsiva sopra riportata, calcola e stampa a su `stdout` il valore di π_n.

Se `n < 0` o se il numero di parametri passati al programma è sbagliato, questo termina con codice `1` senza stampare nulla, in tutti gli altri casi termina con codice `0` dopo aver stampato su `stdout`.

Seguono alcuni esempi:

```
cmd: pigreco 0
stdout: 4.000000

cmd: pigreco 10
```

```
stdout: 3.232316

cmd: pigreco 100
stdout: 3.151493

cmd: pigreco 10000
stdout: 3.141693
```

Non saranno considerate valide soluzioni che non fanno uso della ricorsione per il calcolo di π_n.

Esercizio 2

Una password è composta da n cifre e rispetta i seguenti vincoli:

1. Le cifre, da sinistra a destra, **non decrescono** mai.
2. Esiste **almeno una coppia di numeri consegutivi uguali**.

Seguono esempi di password (valide e non):

- $11111111 - \text{con } n = 8$ rispetta tutti i vincoli;
- $123455 - \text{con } n = 6$ rispetta tutti i vincoli;
- $222 - \text{con } n = 3$ rispetta tutti i vincoli;
- $223450 - \text{con } n = 3$ non rispetta il vincolo sulla lunghezza;
- $123789 - \text{con } n = 10$ non rispetta né il vincolo sulla lunghezza, né il vincolo sulle cifre uguali consecutive.

Nel file `passwords.c` definire la funzione corrispondente alla seguente procedura:

```c
extern void Passwords(int n);
```

La procedura prende in input un numero intero, `n`, e implementa un algoritmo di backtracking che stampa su `stdout` tutte le password di lunghezza `n` che rispettano i vincoli sopra descritti. Il formato della stampa deve corrispondere a quello degli esempi che seguono.

Ad esempio:

- se `n = 1`, la procedura termina senza stampare nulla, essendo impossibile soddisfare il vincolo della cifra consecutiva ripetuta.
- se `n = 2`, la procedura produce il seguente output:

```
1) 00
2) 11
3) 22
4) 33
5) 44
6) 55
7) 66
```

```
 8) 77
 9) 88
10) 99
```

Se `n < 0` la funzione termina senza stampare nulla su `stdout`.

Esercizio 3

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

```
extern Item* Reverse(Item *list, int left);
```

La funzione prende in input una lista di `n` ElemType di tipo `int`, `list`, e un numero intero `left`. La funzione inverte i nodi della lista a partire da quello di indice `left` incluso e restituisce la lista risultante. Si consideri il primo `item` in posizione 0.

Se `left <= 0`, la funzione deve invertire tutti gli elementi della lista.

Se `left >= len(list) - 1`, la funzione ritorna la lista di input, senza modificarla.

La funzione deve modificare il campo `next` degli `Item` della lista di input, non saranno considerate valide soluzione che creano una nuova lista con gli `item` "invertiti" o che modificano il campo `value`.

Seguono alcuni esempi:

```
INPUT:
list = [0, 1, 2, 3, 4, 5, 6, 7, 8], left = 3
OUTPUT
list = [0, 1, 2, 8, 7, 6, 5, 4, 3]
```

```
INPUT:
list = [0, 1, 2, 3, 4, 5, 6, 7, 8], left = 7
OUTPUT
list = [0, 1, 2, 3, 4, 5, 6, 8, 7]
```

```
INPUT:
list = [0, 1, 2, 3, 4, 5, 6, 7, 8], left = 12
OUTPUT
list = [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

```
INPUT:
list = [0, 1, 2, 3, 4, 5, 6, 7, 8], left = 0
```

```
OUTPUT
list = [8, 7, 6, 5, 4, 3, 2, 1, 0]
```

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

```c
typedef int ElemType;

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

e le seguenti funzioni primitive e non:

```c
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 `levelsum.c` definire la funzione corrispondente alla seguente dichiarazione:

```c
extern int LevelSum(const Node *t, size_t level);
```

La funzione `LevelSum()` prende in input un albero binario `t` (di qualunque tipo) e un intero `level` che corrisponde ad un livello dell'albero. Si ricordi che il nodo radice si trova al livello 0.

La funzione deve ritornare la somma di tutti i nodi che si trovano al livello `level`.

Dato ad esempio l'albero `t`:

```
     -2        ---> -2
    /  \
   /    \
  5      1     ---> 6
 / \      \
8   2      1   ---> 11
 \        /
 -9      11    ---> 2
```

- `LevelSum(t, 0)` deve ritornare `-2`;
- `LevelSum(t, 1)` deve ritornare `6`, infatti $5 + 1 = 6$;
- `LevelSum(t, 2)` deve ritornare `11`, infatti $8 + 2 + 1 = 11$.
- `LevelSum(t, 3)` deve ritornare `2`, infatti $-9 + 11 = 2$.
- `LevelSum(t, 4)` deve ritornare `0` dato non ci sono nodi ad altezza 4.

Se `t` è vuoto la funzione deve ritornare sempre 0.

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

```c
typedef int ElemType;

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

e le seguenti funzioni primitive e non:

```c
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

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

```
extern void Stooge(int *vector, size_t vector_size);
```

La funzione prende in input un vettore di interi insieme alla sua dimensione e deve ordinare gli elementi del vettore *in place* utilizzando l'algoritmo *Stooge Sort*.

Stooge Sort è un algoritmo di ordinamento ricorsivo dalla complessità temporale eccezionalmente negativa di $\mathcal{O}(n^{log(3)/log(1.5)}) = \mathcal{O}(n^{2.7095...})$. Il tempo di esecuzione dell'algoritmo è quindi maggiore anche rispetto al *Bubble Sort*, un esempio canonico di ordinamento abbastanza inefficiente.

L'algoritmo Stooge Sort può essere riassunto come segue:

1. se il primo elemento è maggiore dell'ultimo, scambiare i due elementi;
2. **se il vettore è composto da tre o più elementi**, ordinare i primi $\frac{2}{3}$, poi gli ultimi $\frac{2}{3}$, infine di nuovo i primi $\frac{2}{3}$ del vettore **usando lo stooge sort ricorsivamente**.

In tutti i casi, i $\frac{2}{3}$ vanno arrotondati per eccesso, ad esempio se il vettore è composto da 7 elementi, $\lceil 7 * \frac{2}{3} \rceil = 5$.