Esame di Laboratorio del 18/01/2023

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

Un numero n si dice triangolare se è rappresentabile in forma di triangolo, ovvero, preso un insieme con una cardinalità n è possibile disporre i suoi elementi su una griglia regolare, in modo da formare un triangolo equilatero. I numeri 1, 3, 6, 10 sono ad esempio triangolari, infatti:

```
1    3      6       10
*    *      *        *
    * *    * *      * *
          * * *    * * *
                  * * * *
```

La sequenza dei numeri triangolari a partire da 0 è la seguente:

```
0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, ...
```

Questa sequenza è definita dalla relazione ricorsiva:

$$S_0 = 0 \tag{1}$$
$$S_1 = 1 \tag{2}$$
$$S_2 = 3 \tag{3}$$
$$S_n = 3S_{N-1} - 3S_{N-2} + S_{N-3} \tag{4}$$

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

```
tnumbers <n>
```

Il programma prende in input un numero intero positivo `n`, e, utilizzando la funzione ricorsiva sopra riportata, calcola e stampa a video l'n-esimo numero triangolare.

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: tnumbers 0
stdout: 0

cmd: tnumbers 1
stdout: 1

cmd: tnumbers 12
stdout: 78

cmd: tnumbers 17
stdout: 153
```

Non saranno considerate valide soluzioni che non fanno uso della ricorsione per determinare S_N.

Esercizio 2

In un *orologio binario*, sia le ore che i minuti sono rappresentati come somma di potenze del 2. Sul display sono presenti 5 led per le ore (nella riga superiore) e 6 led per i minuti (nella riga inferiore).

Ad esempio, l'orologio in figura segna le ore 17:15. Infatti, nella riga delle ore sono accesi i led in corrispondenza dei valori 16 e 1, la cui somma è 17, mentre nella riga dei minuti sono accesi 8, 4, 2, e 1, la cui somma è 15. In questo caso, sono accesi in totale 6 led.

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

```
extern void ValidTimes(uint8_t n);
```

La procedura prende in input un numero intero senza segno, `n`, e implementa un algoritmo di backtracking che **stampa su `stdout` tutti i possibili orari rappresentabili con esattamente `n` led accesi** su un orologio binario.

Devono essere stampati **solo orari validi**, ovvero le cui ore siano al massimo 23 e i minuti al massimo 59.

Ad esempio, se `n = 1` la procedura deve stampare (non necessariamente in quest'ordine):

```
00:01, 00:02, 00:04, 00:08, 00:16, 00:32,
01:00, 02:00, 04:00, 08:00, 16:00,
```

Invece se `n = 9`, l'output deve essere (non necessariamente in quest'ordine):

```
15:31, 15:47, 15:55, 15:59, 23:31, 23:47, 23:55, 23:59,
```

Il formato dell'output deve corrispondere a quello degli esempi, ovvero `hh:mm`. Sia le ore che i minuti devono essere stampati con 2 cifre. Gli orari sono separati dai caratteri `','` e `' '`.

Se `n > 11`, o se non esiste nessun orario valido, la funzione non deve stampare nulla.

Esercizio 3

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

```
extern Item* ShiftN(Item *list, size_t n);
```

La funzione prende in input una lista di ElemType di tipo `int`, `list`, e un numero `n`. La funzione sposta in avanti di `n` posizioni il primo nodo della lista. Se `n >= len(list)` la funzione deve comportarsi **come se la lista fosse circolare**, ovvero come se l'ultimo elemento puntasse al primo. Se la lista è vuota o contiene un solo elemento, la funzione ritorna la lista 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 o che modificano il campo `value`.

Seguono alcuni esempi:

```
INPUT:
list = [0, 1, 2, 3], n = 2
```

```
step 1: [1, 0, 2, 3]
step 2: [1, 2, 0, 3]

OUTPUT
list = [1, 2, 0, 3]
```

```
INPUT:
list = [0, 1, 2, 3], n = 3

step 1: [1, 0, 2, 3]
step 2: [1, 2, 0, 3]
step 3: [1, 2, 3, 0]

OUTPUT
list = [1, 2, 3, 0]
```

```
INPUT:
list = [0, 1, 2, 3], n = 4

step 1: [1, 0, 2, 3]
step 2: [1, 2, 0, 3]
step 3: [1, 2, 3, 0]
step 4: [1, 0, 2, 3]

OUTPUT
list = [1, 0, 2, 3]
```

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

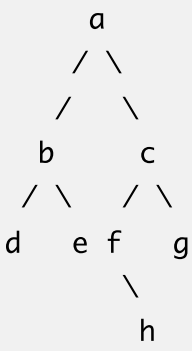
Nel file `ancestor.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern const Node* LowestCommonAncestor(const Node *t,
                                        const Node *n1,
                                        const Node *n2);
```

La funzione `LowestCommonAncestor()` prende in input un albero binario `t` (di qualunque tipo) e due nodi appartenenti ad esso, `n1` e `n2`.

La funzione deve cercare e ritornare il "più basso" antenato comune a `n1` e `n2` in `t`. In altre parole, la funzione deve trovare il più piccolo sottoalbero di `t` contenente sia `n1` che `n2`.

Dato ad esempio l'albero `a`:



- `LowestCommonAncestor(a, h, g)` deve ritornare `c`, in quanto il sottoalbero identificato da `c` è il più piccolo albero che contiene sia `h` che `g`.
- `LowestCommonAncestor(a, d, f)` deve ritornare `a`, in quanto non esiste un sottoalbero di `a` che contenga sia `d` che `f`. Si noti che `t` è sempre un antenato comune a `n1` e `n2`, ma non è detto che sia il "più basso".

- `LowestCommonAncestor(a, e, e)` deve ritornare `e`.
- `LowestCommonAncestor(a, h, f)` deve ritornare `f`.

Si può assumere che `t` sia sempre non vuoto e che i nodi `n1` e `n2` facciano sempre parte di `t`.

Suggerimento: si consiglia di implementare la funzione ausiliaria `bool TreeContainsNode(const Node* tree, const Node* node)`, che dato un albero e un nodo, ritorna `true` se il nodo appartiene all'albero, `false` altrimenti.

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

```
typedef char 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

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

```
extern int LastStoneWeight(Heap *h);
```

La funzione prende in input un max-heap contenente i pesi di `h->size` pietre, utilizzate per un gioco. Ad ogni turno vengono scelte le due pietre più pesanti (il cui peso è rispettivamente `x` e `y` con `x <= y`) e vengono sbattute l'una con l'altra. Il risultato dello scontro può essere:

- Se `x == y`, entrambe le pietre vengono distrutte;
- Se `x != y`, la pietra di peso `x` viene distrutta, mentre quella di peso `y` viene rimessa nel mucchio con il nuovo peso `y - x`.

Il gioco finisce quando nel mucchio di pietre non ci sono più pietre o ne rimane una sola. La funzione deve restituire 0 nel primo caso, il peso dell'ultima pietra rimasta nel secondo.

Si può assumere che ci sia sempre almeno una pietra nel mucchio.

Data ad esempio l'heap:

```
  77
 /  \
21  18
```

la funzione deve ritornare 38.

La procedura `Pop()` potrebbe tornare utile. Data una max-heap, la procedura ritorna (tramite il parametro di output `popped`) il nodo del valore massimo dopo averlo rimosso dall'heap. La `Pop` preserva le proprietà max-heap.

```
void Pop(Heap* h, ElemType *popped) {
    if (h->size == 0) {
        popped = NULL;
        return;
    }

    *popped = ElemCopy(&h->data[0]);
    ElemSwap(&h->data[0], &h->data[h->size - 1]);
    h->size--;
    h->data = realloc(h->data, sizeof(ElemType) * h->size);
    HeapMaxMoveDown(h, 0);
    return;
}
```

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

```
typedef int ElemType;

struct Heap{
    ElemType *data;
    size_t size;
};
typedef struct Heap Heap;
```

e le seguenti funzioni primitive e non:

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

int LeftHeap(int i);
int RightHeap(int i);
int ParentHeap(int i);
Heap* CreateEmptyHeap();
void InsertNodeMinHeap(Heap *h, const ElemType *e);
bool IsEmptyHeap(const Heap *h);
ElemType* GetNodeValueHeap(const Heap *h, int i);
void MoveUpMinHeap(Heap *h, int i);
void MoveDownMinHeap(Heap *h, int i);
void DeleteHeap(Heap *h);
void WriteHeap(const Heap *h, FILE *f);
void WriteStdoutHeap(const Heap *i);
```

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