Esame di Laboratorio del 18/01/2022

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.

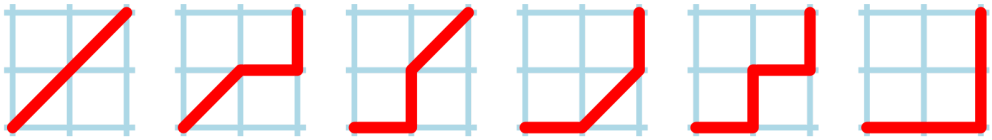
Esercizio 1

In matematica, data una griglia quadrata di dimensione $n \times n$ nel 1° quadrante di un sistema di riferimento cartesiano, il numero di Schröder, S_n, descrive il numero di cammini possibili per arrivare dal punto di coordinate (0, 0) al punto di coordinate (n, n), ammettendo di potersi muovere soltanto verso l'alto (Nord), verso destra (Est) o lungo la diagonale NE, e senza che il cammino oltrepassi mai la bisettrice del primo quadrante, data dalla retta di equazione $y = x$.

La successione di tali numeri interi, che prendono il nome dal matematico tedesco Ernst Schröder, per $n = 0, 1, \ldots$, ha come primi elementi:

$$1, 2, 6, 22, 90, 394, 1806, 8558, \ldots$$

La figura seguente mostra i 6 cammini che, rispettando le sopraccitate condizioni, è possibile fare per raggiungere il punto di coordinate $(2, 2)$ partendo dal punto di coordinate $(0, 0)$.



[Fonte: Wikipedia]

Una delle possibili relazioni di ricorrenza che descrive la successione dei numeri di Schröder è la seguente:

$$S_n = \begin{cases} 1, & \text{se } n = 0 \\ 2, & \text{se } n = 1 \\ \frac{(6n-3)\cdot S_{n-1}}{n+1} - \frac{(n-2)\cdot S_{n-2}}{n+1}, & \text{se } n \geq 2 \end{cases}$$

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

```
schroeder <n>
```

Il programma `schroeder` prende in input un numero numero `n` e stampa a video (`stdout`) l'n-esimo termine delle successione di Schröder, calcolato **ricorsivamente**.

Se il numero dei parametri passati al programma non è corretto o se `n < 0`, questo termina con codice `1`, senza stampare nulla, altrimenti termina con codice di uscita `0` dopo aver completato la stampa.

Non saranno considerate valide soluzioni che non fanno uso della ricorsione per calcolare l'n-esimo numero della successione.

Esercizio 2

Nel file `password.c` implementare la definizione della funzione corrispondente alla seguente dichiarazione:

```
extern int Password(const char *str, int n);
```

Data una stringa C contenente caratteri alfanumerici, la funzione deve stampare a video (`stdout`) tutte le possibili combinazioni di caratteri (password) di lunghezza `n` che è possibile costruire utilizzando i caratteri a disposizione, **eventualmente ripetuti**. Si utilizzi a tale scopo un algoritmo di **backtracking**.

Ogni parola deve essere separata dalla precedente dal carattere `<a capo>`. L'ordine con cui vengono visualizzate le soluzioni non è significativo.

La funzione ritorna il numero di soluzioni trovate. Se la stringa è vuota la funzione ritorna 0 senza stampare nulla.

Data ad esempio la stringa `"a1"` e `n = 1`, la funzione deve stampare (non necessariamente in questo ordine):

```
a
1
```

e ritornare 2.

Se `n = 2` la funzione deve invece stampare:

```
aa
a1
1a
11
```

e ritornare 4.

Esercizio 3

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

```
extern Item *PariDispari(Item *i);
```

La funzione prende in input una lista di interi, `i`, e la **modifica** in modo che tutti i valori pari siano prima di quelli dispari. L'ordine in cui i valori compaiono non è importante, purché tutti i valori pari siano prima di quelli dispari. La funzione ritorna quindi la lista risultante. La funzione deve avere complessità computazionale O(n) e un costo di memoria pari a O(1).

In sostanza, la funzione deve modificare il campo `next` degli `Item` esistenti, senza allocare nuova memoria. Non saranno considerate valide soluzioni che producono una *nuova* lista per ottenere il risultato richiesto. Non è consentito modificare i `value` degli `Item`.

Se la lista di input è vuota la funzione ritorna `NULL`.

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.

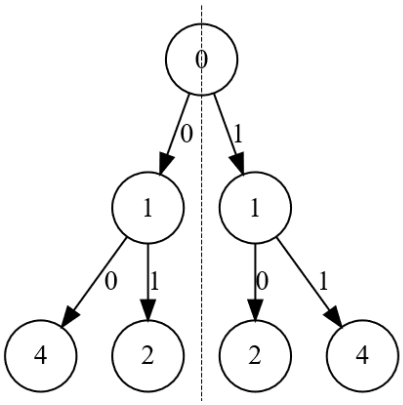
Esercizio 4

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

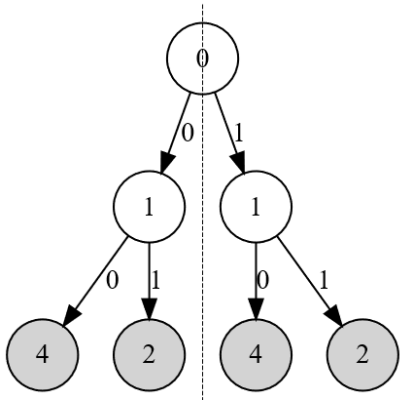
```
extern bool TreeIsMirror(Node *t);
```

La funzione prende in input un albero binario di `int` e deve ritornare `true` se questo è specchio di sé stesso, ovvero se è simmetrico rispetto all'asse centrale, `false` altrimenti. Alberi vuoti o contenenti solamente la radice sono da considerarsi simmetrici.

Ad esempio, l'albero:



è specchio di sé stesso e quindi la funzione deve ritornare `true`, mentre l'albero:



non lo è (in grigio i nodi che non rispettano la proprietà di simmetria) e quindi la funzione deve ritornare `false`.

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 *gnome sort*, denominato anche *stupid sort*, è un algoritmo molto semplice che non richiede cicli nidificati. Il nome deriva dal modo con cui uno gnomo da giardino ordina una fila di vasi da fiori, nello specifico:

- Lo gnomo guarda il vaso di fiori accanto a sé e quello precedente. Se i vasi sono nell'ordine giusto lo gnomo avanza al vaso successivo, altrimenti li scambia e fa un passo indietro al vaso precedente.
- Se non c'è un vaso precedente (lo gnomo è all'inizio del giardino), lo gnomo fa un passo avanti; se non c'è alcun vaso accanto a sé (lo gnomo è alla fine del giardino), ha terminato l'ordinamento.

In altre parole, i passi dell'algoritmo possono essere riassunti come segue:

1. Se sei all'inizio del vettore, vai all'elemento successivo, ovvero ti sposti dalla posizione 0 alla posizione 1;
2. Se l'elemento corrente, `i`, del vettore è maggiore o uguale all'elemento precedente, `i - 1`, fai un passo avanti portandoti all'indice `i + 1`;
3. Se l'elemento corrente, `i`, del vettore è minore dell'elemento precedente, scambia questi due elementi e fai un passo indietro portandoti all'indice `i - 1`;
4. Ripetere i passaggi 2. e 3. fino a quando `i` raggiunge la fine dell'array.

Sia dato ad esempio il vettore `v = [3, 6, 2, 4]`, i passi dell'algoritmo sono i seguenti (ad ogni passo la freccia rappresenta la posizione corrente dello gnomo):

```
    ↓
01) 3, 6, 2, 4

       ↓
02) 3, 6, 2, 4

          ↓
03) 3, 6, 2, 4

       ↓
04) 3, 2, 6, 4
```

```
    ↓
05) 2, 3, 6, 4

       ↓
06) 2, 3, 6, 4

          ↓
07) 2, 3, 6, 4

             ↓
08) 2, 3, 6, 4

          ↓
09) 2, 3, 4, 6

             ↓
10) 2, 3, 4, 6

                ↓
11) 2, 3, 4, 6
```

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

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

La funzione implementa l'algoritmo di ordinamento *gnome sort* appena descritto.