Esame di Laboratorio del 21/07/2022

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:

```
power4 <n>
```

Il programma prende in input un numero intero positivo o al più nullo, `n`, e deve stampare a video `4^x = n` (dove `n` ed `x` devono essere sostituiti rispettivamente dal valore di input e dalla soluzione al problema) se il numero è una potenza del 4, `!p4` altrimenti. Internamente, deve essere utilizzata una funzione **ricorsiva** per determinare se `n` è o meno una potenza del 4.

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

Seguono alcuni esempi:

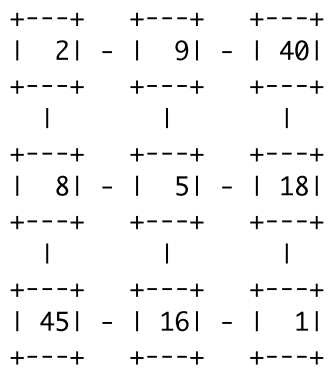
```
cmd: power4 4
stdout: 4^1 = 4

cmd: power4 16
stdout: 4^2 = 16

cmd: power4 8
stdout: !p4
```

Non saranno considerate valide soluzioni che non fanno uso della ricorsione per determinare se `n` è o meno un multiplo di 4.

Esercizio 2

Un gioco della settimana enigmistica chiede di riempire le celle di una griglia 3x3 con dei numeri dati, affinché il prodotto di tre numeri allineati (su righe o colonne) sia costante.

Data la sequenza di numeri s = {1, 2, 5, 8, 9, 16, 18, 40, 45}, un modo per riempire la griglia rispettando i vincoli sui prodotti è rappresentato di seguito:



In questo esempio il prodotto dei numeri allineati è 720.

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

```
extern int* RisolviProdotto(int n, const int *s);
```

La funzione prende in input un numero interno, `n`, che rappresenta la dimensione del lato della griglia da popolare e un vettore di numeri la cui dimensione è sempre `n * n`. La funzione deve, utilizzando un algoritmo di *backtracking*, individuare uno dei modi con cui è possibile riempire la griglia utilizzando (tutti e soli) i numeri in `s` e rispettando i vincoli, ovvero garantendo che il prodotto di `n` numeri allineati sia sempre costante. I numeri in `s` sono sempre < 1000 e > -100. Al termine, la funzione ritorna un vettore allocato dinamicamente che rappresenta la soluzione trovata, memorizzata per righe. Il vettore soluzione dovrà avere dimensione pari a `n*n`. Se non esistono soluzioni valide la funzione ritorna `NULL`.

Si può assumere che i parametri di input della funzione `RisolviProdotto()` siano sempre corretti, ovvero `n > 0` ed `s != NULL` e con dimensione pari a `n * n`.

Esercizio 3

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

```
extern ElemType* NextGreater(const Item *list, size_t *answer_size);
```

La funzione prende in input una lista di `n` ElemType di tipo `int`, `list`, e ritorna un vettore allocato dinamicamentimente, `answer`, di `n` elementi dello stesso tipo. Al termine della funzione, il parametro di output `answer_size` dovrà contenere la dimensione di `answer` (uguale ad `n`).

Numeriamo gli elementi di `list` da `0` a `n - 1`. La funzione deve scorrere la lista e per ogni elemento scrivere in `answer[i]` il valore del successivo nodo maggiore. In altre parole, per ogni nodo la funzione trova e scrive nella posizione corrispondente del vettore `answer` il valore del primo nodo successivo strettamente maggiore. Se l'i-esimo elemento non è seguito da nodi maggiori si scriva `INT_MIN` dentro `answer[i]`.

La lista di input può essere vuota, in questo caso `answer_size` dovrà essere `0` e il valore di ritorno della funzione `NULL`.

Data ad esempio la lista `list = [2, 3, 1, 4, 7, 2]` la funzione ritorna il vettore `answer = {3, 4, 4, 7, INT_MIN, INT_MIN}` dove il valore di `INT_MIN` dipende dall'architettura su cui il codice viene compilato. Infatti, il primo valore più grande (e successivo) di 2 è 3, il primo valore più grande (e successivo) di 3 è 4, il primo valore più grande (e successivo) di 1 è 4, 7 non è seguito da valori più grandi e lo stesso vale per il 2 finale. Si noti che l'ultimo elemento non sarà mai seguito da numeri più grandi, quindi l'ultimo elemento di `answer` varrà sempre `INT_MIN`.

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

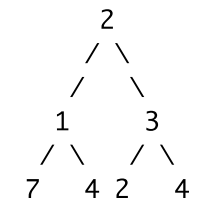
Nel file `invert.c` definire la funzione corrispondente alla seguente dichiarazione:

```c
extern Node* Invert(Node *t);
```

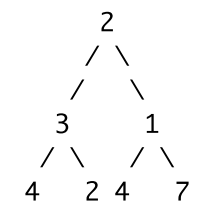
La funzione `Invert()` prende in input un albero binario, `t`, e lo inverte ritornando l'albero risultante. Invertire un albero consiste nello scambiare di posto i figli di ciascun nodo non foglia.

La funzione non deve creare un nuovo albero, bensì modificare quello esistente, e ritornarne il puntatore alla radice. Se l'abero di input è vuoto, la funzione ritorna un albero vuoto.

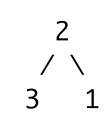
Dato ad esempio l'albero:



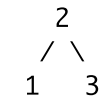
la funzione deve ritornare:



Dato invece l'albero:



la funzione ritorna:



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

Il problema della *superstringa più corta*, o *shortest superstring* in inglese, prende come input stringhe di diversa lunghezza (dizionario) e trova la stringa più corta che contiene tutte le stringhe di input come sottostringhe.

Per risolvere il problema è possibile utilizzare un algoritmo greedy che ad ogni passo "unisce" alla stringa di output la stringa nel dizionario con sovrapposizione maggiore di caratteri e ripete fino a quando tutte le stringhe sono state unite. Nel caso in cui più stringhe abbiano lo stesso grado di sovrapposizione basta sceglierne una a caso.

Inizialmente la stringa di output è vuota, quindi occorre scegliere la coppia di parole con sovrapposizione maggiore. Anche in questo caso, se più coppie hanno la stessa sovrapposizione si sceglie una coppia casuale, ad esempio la prima o l'ultima trovata.

Sia dato ad esempio il seguente dizionario:

```
dizionario = {"ABCDH", "DHE", "BCDH", "HEF"}
```

Al primo passo devo scegliere la coppia con sovrapposizione maggiore di caratteri:

```
("ABCDH","DHE")  = "ABCDHE"   2 caratteri sovrapposti
                      **

("ABCDH","BCDH") = "ABCDH"    4 caratteri sovrapposti
                    ****

("ABCDH","HEF")  = "ABCDHEF"  1 carattere sovrapposto
                       *

("DHE","BCDH")   = "BCDHE"    2 caratteri sovrapposti
                      **

("DHE","HEF")    = "DHEF"     2 caratteri sovrapposti
                      **

("BCDH","HEF")   = "BCDHEF"   1 carattere sovrapposto
                       *
```

quindi sceglierò la coppia `("ABCDH", "BCDH")` che produrrà in output la stringa `"ABCDH"`.

Al secondo passo devo scegliere tra le parole rimanenti, ovvero, `{"DHE", "HEF"}`, quella che ha la maggior sovrapposizione con`"ABCDH"`:

```
("ABCDH","DHE")  = "ABCDHE"   2 caratteri sovrapposti
                      **

("ABCDH","HEF")  = "ABCDHEF"  1 carattere sovrapposto
                       *
```

Scelgo la prima possibilità, ottenendo `"ABCDHE"`.

Ripetendo il procedimento si otterrà la stringa `"ABCDHEF"`.

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

```
extern char* SolveSuperstring(const char **v, size_t v_size);
```

La funzione prende in input un vettore di stringhe C zero terminate, `v`, e la sua dimensione, `v_size`, e deve applicare l'algoritmo greedy sopra descritto per determinare la superstringa "più corta" contenente tutte le stringhe di `v` come sottostringhe.

La funzione ritorna una stringa C zero terminata e allocata dinamicamente contenente la soluzione.

Si noti che essendo l'algoritmo descritto greedy è possibile che esistano soluzioni migliori di quella ritornata dalla funzione `SolveSuperstring()`.

Per la risoluzione di questo esercizio potete fare uso della funzione `Overlap` che date due stringhe C zero terminate assegna al paramtro di output `overlapping` il numero di caratteri di massima sovrapposizione e ritorna il puntatore ad una nuova stringa C allocata dinamicamente contenente il risultato della sovrapposizione.

```
#define _CRT_SECURE_NO_WARNINGS

#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

static char* Overlap(const char* str1, const char* str2, int* overlapping) {

    const int len1 = (int)strlen(str1);
    const int len2 = (int)strlen(str2);
    const int lenShort = len1 < len2 ? len1 : len2;
    const int lenLong = len1 < len2 ? len2 : len1;
    const char* strShort = len1 < len2 ? str1 : str2;
    const char* strLong = len1 < len2 ? str2 : str1;

    // Check if strShort is a substring of strLong
    if (strstr(strLong, strShort) != NULL) {
        char* res = malloc(lenLong + 1);
        strcpy(res, strLong);
        *overlapping = lenShort;
        return res;
    }

    int maxOverlap = 0;
    bool longGoesFirst = true;

    // Try to put strShort at the end of strLong
    for (int i = lenShort - 1; i > 0; --i) {
        if (strncmp(strLong + lenLong - i, strShort, i) == 0) {
            maxOverlap = i;
            break;
        }
    }

    // Try to put strLong and the end of strShort
    for (int i = lenShort - 1; i > maxOverlap; --i) {
        if (strncmp(strLong, strShort + lenShort - i, i) == 0) {
            if (i > maxOverlap) {
                maxOverlap = i;
                longGoesFirst = false;
```

```
            }
            break;
        }
    }

    char* res = malloc(lenLong + lenShort - maxOverlap + 1);
    if (longGoesFirst) {
        strcpy(res, strLong);
        strcpy(res + lenLong, strShort + maxOverlap);
    }
    else {
        strcpy(res, strShort);
        strcpy(res + lenShort, strLong + maxOverlap);
    }
    *overlapping = maxOverlap;
    return res;
}
```