# Esame di Laboratorio del 18/02/2022

## 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

La mappa logistica è un modello demografico che viene utilizzato per descrivere due effetti:

- *riproduzione*, ovvero crescita della popolazione ad un tasso proporzionale a quella corrente quando la popolazione iniziale è piccola;
- *mortalità*, ovvero che il tasso di crescita diminuisce con una velocità proporzionale alla differenza tra la "portata" teorica dell'ambiente (risorse) e la popolazione corrente.

In formula:

$$x_{n+1} = r x_n (1 - x_n)$$

dove:

$x_n$ è un numero compreso tra 0 e 1, e rappresenta il rapporto tra la popolazione esistente e quella massima possibile in un anno $n$. $x_0$ rappresenta quindi il rapporto tra la popolazione iniziale (all'anno 0) e quella massima. $r$ è un numero positivo e rappresenta il tasso combinato tra la riproduzione e la mortalità.

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

```
logisticmap <x0> <r> <n>
```

Il programma `logisticmap` prende in input due numeri con la virgola, $x_0$ e $r$, e un numero intero `n`. Il programma deve stampare a video (`stdout`) l'n-esimo termine della mappa logistica calcolato **ricorsivamente**.

Se il numero dei parametri passati al programma non è corretto, o se `x0 < 0`, `x0 > 1`, `r < 0` o `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 valore della mappa logistica.

Ad esempio, il comando

```
logisticmap 0.1 3 5
```

deve stampare `0.721109`.

## Esercizio 2

Bob vuole costruire un impero di $n$ città, numerate da $c_0$ a $c_{n-1}$. Ogni minuto, da ogni città $c_i, i \in [0, n-2]$, parte un treno verso $c_{i+1}$. Ogni città di partenza ha il proprio treno, che può portare un certo numero di persone $t_i$.

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

```
int TempoTrasporto(int n, const int* t, int p);
```

Considerato un impero di $n$ città, la funzione ritorna il tempo minimo necessario, in minuti, per trasportare $p$ persone dalla città $c_0$ alla città $c_{n-1}$. Il parametro $t$ è un vettore di $n-1$ interi, contenente all'elemento $t[i]$ la capienza del treno che viaggia da $c_i$ a $c_{i+1}$. Si trascuri il tempo per i cambi. I parametri passati alla funzione saranno sempre corretti.

Esempio:

```
n = 4
t = {3, 2, 4}
p = 5
```

Vediamo, minuto per minuto, quante persone ci sono in ciascuna città:

| | $m_0$ | $m_1$ | $m_2$ | $m_3$ | $m_4$ | $m_5$ |
|---|---|---|---|---|---|---|
| $c_0$ | 5 | 2 | | | | |
| $c_1$ | | 3 | 3 | 1 | | |
| $c_2$ | | | 2 | 2 | 1 | |
| $c_3$ | | | | 2 | 4 | 5 |

- All'istante iniziale, il minuto $m_0$, tutte le 5 persone sono nella città $c_0$.
- Il treno $c_0 \to c_1$ ha una capienza di 3 persone, quindi al tempo $m_1$ avrà trasportato 3 delle 5 persone a $c_1$.
- Al tempo $m_2$, il treno $c_1 \to c_2$ avrà portato 2 persone in $c_2$. Contemporaneamente, però,

saranno arrivate altre 2 persone da $c_0$ a $c_1$, riportando il totale di passeggeri in $c_1$ a 3.

- ...
- Al minuto 5, tutte le persone si trovano nella città finale; la funzione dovrà quindi ritornare il valore $5$.

Per la risoluzione di questo esercizio occorre utilizzare la tecnica greedy, ovvero effettuare ad ogni passo la scelta più appetibile.

## Esercizio 3

Creare i file `remove.h` e `remove.c` che consentano di utilizzare la seguente funzione:

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

La funzione prende in input una lista di elementi interi, `i`, e rimuove gli elementi duplicati senza allocare nuova memoria e con complessità O(n). La lista di input è **sempre ordinata** in senso crescente o decrescente. Ovviamente, per avere complessità O(n) l'implementazione deve sfruttare l'ordinamento.

La funzione ritorna la lista ottenuta dopo la rimozione dei duplicati o `NULL` se la lista di input è `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 `read.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern Node *TreeRead(const char *filename);
```

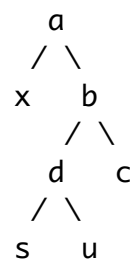
La funzione prende in input il nome di un file di testo, `filename`, che contiene i dati di un albero binario di caratteri. I dati all'interno del file sono così strutturati:

- Ogni carattere alfanumerico identifica un nodo dell'albero;
- Le foglie sono precedute dal carattere `'.'`;
- Ogni nodo è seguito dal figlio sinistro (che a sua volta può avere figli) e poi da quello destro (che a sua volta può avere figli);
- Ogni nodo, ad eccezione delle foglie, ha sempre entrambi i figli;
- I *whitespace* (`' '`, `'\t'`, `'\r'`, `'\n'`, `'\v'`, `'\f'`) all'interno del file non hanno alcun significato.

Dato ad esempio il file:

```
a
    .x
    b
        d
            .s
            .u
        .c
```

la funzione deve produrre l'albero:




Dal momento che i *whitespace* non hanno significato, il file che segue corrisponde anch'esso all'albero appena mostrato:

```
a.xbd.s.u.c
```

La funzione deve aprire il file in modalità lettura tradotta, leggerne il contenuto e costruire l'albero corrispondente che deve quindi essere ritornato. Si assuma che il file di input sia sempre correttamente formato, ovvero che rispetti sempre la sintassi precedentemente definita. Se non è possibile aprire il file o se il file è vuoto la funzione deve ritornare un albero vuoto (`NULL`).

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

L'algoritmo di ordinamento *counting sort* è un algoritmo di ordinamento per valori numerici interi che non richiede cicli nidificati. In sostanza, l'algoritmo consiste nel contare il numero di occorrenze di ciascun valore presente nel vettore da ordinare, memorizzando l'informazione in un array temporaneo di dimensione pari all'intervallo di valori. In termini operativi, dato il vettore `v` l'algoritmo può essere riassunto come segue:

1. si determinano i valori massimo, `max(v)`, e minimo, `min(v)` del vettore e si alloca memoria per un vettore `tmp` di `max(v) - min(v) + 1` elementi. Questo vettore rappresenta la frequenza di ogni elemento in `v`, nello specifico, in `tmp[i]` avremo la frequenza di `i + min(v)`;
2. si visita il vettore `v` in ordine e si aggiornano opportunamente le frequenze in `tmp`;
3. si visita il vettore `tmp` in ordine e si scrivono nel vettore di input, `v`, `tmp[i]` copie del valore `i+min(v)`.

Ad esempio, dato `v = { 3, 4, 3, 4, 5, 7, 5, 5, 3 }`

1. si determinano `max(v) = 7` e `min(v) = 3` e si alloca un vettore `tmp` di `7 - 3 + 1 = 5` elementi;
2. si itera su `v` aggiornando le frequenze in `tmp`, al termine avremo `tmp = { 3, 2, 3, 0, 1 }`, infatti, in `v` sono presenti tre 3, due 4, tre 5, zero 6 e un 7;
3. si itera su `tmp` scrivendo `tmp[i]` volte il valore `i + min(v)`, quindi tre volte 3, due volte 4 e via così. Al termine avremo `v = { 3, 3, 3, 4, 4, 5, 5, 5, 7 }`

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

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

La funzione implementa l'algoritmo di ordinamento *counting sort* appena descritto, ordinando il vettore `v` di `v_size` elementi.