

# Esame di Tecniche di Programmazione del 17/06/2016

---

## Esercizio 1

Sia data la seguente struttura Mat:

```
typedef struct Mat {  
  
    int cols; // numero di colonne  
  
    int rows; // numero di righe  
  
    float** row_ptrs; // puntatori alle righe  
  
} Mat;
```

Implementare la seguente funzione C:

```
void matrix_compute_neighboor_sub(Mat* dest, Mat* src);
```

che dati in input:

- un puntatore ad una matrice dest, preallocata ed avente le stesse dimensioni di src
- un puntatore ad una matrice src

Inserisca in ogni cella  $[r, c]$  della matrice dest, la differenza tra la cella  $[r, c]$  e tutti i suoi otto vicini.

Gli otto vicini della cella  $[r,c]$  sono le celle  $[r-1, c-1]$ ,  $[r-1, c]$ ,  $[r-1, c+1]$ ,  $[r, c-1]$ ,  $[r, c+1]$ ,  $[r+1, c-1]$ ,  $[r+1, c]$  e  $[r+1, c+1]$ . Se una cella si trova sul bordo, i "vicini" di una cella non includono elementi fuori della matrice.

## Esempio

Data una struttura Mat src contenente la seguente matrice 5x5:

```
3 4 3 3 4 2  
4 4 1 0 -3 3  
3 -4 1 -2 1 -2
```

e data una struttura Mat dest contenente una matrice della stessa dimensione di src, richiamando la seguente linea di codice:

```
matrix_compute_neighboor_sub(&dest, &src);
```

si ottiene il seguente effetto su dest:

```
-9 -11 -9 -2 -1 -2  
-6 -11 -8 -8 -12 1
```

```
#include <math.h>

#include "esercizio1.h"

void matrix_compute_neighboor_sub(Mat* dest, Mat* src) {
    for(int r = 0; r < src->rows; ++r) {
        for(int c = 0; c < src->cols; ++c) {
            float sum = 0.0f;
            for(int rr = r - 1; rr <= r + 1 && rr < src->rows; ++rr) {
                for(int cc = c - 1; cc <= c + 1 && cc < src->cols; ++cc) {
                    if(rr >= 0 && cc >= 0 && !(rr == r && cc == c)) {
                        sum += src->row_ptrs[rr][cc];
                    }
                }
            }
            dest->row_ptrs[r][c] = src->row_ptrs[r][c] - sum;
        }
    }
}
```

---

## Esercizio 2

Sia data una lista i cui elementi appartengono alla seguente struttura ListNode:

```
typedef struct ListNode {

    int value;

    struct ListNode* next;

} ListNode;
```

Si realizzi la funzione C:

```
ListNode* merge_elements(ListNode* l1, ListNode* l2)
```

che date due liste l1 e l2 **ordinate** in senso crescente in base al campo value, restituisca in output una nuova lista. Tale lista deve contenere tutti gli elementi di l1 ed l2, ed essere anche essa **ordinata** in senso crescente in base al campo value.

### Esempio

Data la seguente lista l1:

-13 -> 0 -> 16 -> 27 -> 29

e data la lista l2:

3 -> 4 -> 23

Richiamando la seguente linea di codice:

```
ListNode* l_merge = merge_elements(&l1, &l2)
```

l\_mrge dovrà contenere:

-13 -> 0 -> 3 -> 4 -> 16 -> 23 -> 27 -> 29

```
#include <stdlib.h>

#include "esercizio2.h"

ListNode* merge_elements(ListNode* l1, ListNode* l2) {
    ListNode* gen_node = (ListNode*)calloc(1, sizeof(ListNode));
    gen_node->next = NULL;
    ListNode* list = gen_node;
    while(l1 != NULL || l2 != NULL) {
        list->next = (ListNode*)calloc(1, sizeof(ListNode));
        list = list->next;
        list->next = NULL;
        if(l1 != NULL && l2 == NULL) {
            list->value = l1->value;
            l1 = l1->next;
        }
        else if(l2 != NULL && l1 == NULL) {
            list->value = l2->value;
            l2 = l2->next;
        }
    }
}
```

```

else {

    if(l1->value < l2->value) {

        list->value = l1->value;

        l1 = l1->next;

    }

    else {

        list->value = l2->value;

        l2 = l2->next;

    }

}

}

list = gen_node->next;

free(gen_node);

return list;

}

```

---

## Esercizio 3

Con riferimento alla struttura collegata lineare descritta di seguito:

```

typedef struct ListNode {

    int value;

    struct ListNode* next;

} ListNode;

```

implementare la seguente funzione **ricorsiva** C (nessuna istruzione di ciclo è permessa):

```
ListNode* list_multipliers(ListNode* src, int n);
```

che data una lista src restituisca una nuova lista contenente gli elementi di src che sono multipli di n.

## Esempio

Data la seguente lista src:

-9 -> 9 -> -6 -> 7 -> 8 -> -6 -> 5 -> 0 -> 3 -> -4

Richiamando la seguente linea di codice:

```
ListNode* l_int = list_multipliers(&src, 4);
```

l\_int dovrà contenere:

8 -> 0 -> -4

```
#include "esercizio3.h"

#include <stdlib.h>

ListNode* list_multipliers(ListNode* src, int n) {
    if(src == NULL) {
        return NULL;
    }
    else if((src->value % n) == 0) {
        ListNode *list = list_multipliers(src->next, n);
        ListNode *node = (ListNode*)malloc(sizeof(ListNode));
        node->value = src->value;
        node->next = list;
        return node;
    }
    else {
        return list_multipliers(src->next, n);
    }
}
```

---

# Esame di Tecniche di Programmazione del 08/07/2016 - A

---

## Esercizio 1

Sia data la seguente struttura Mat:

```
typedef struct Mat {  
  
    int cols; // numero di colonne  
  
    int rows; // numero di righe  
  
    float** row_ptrs; // puntatori alle righe  
  
} Mat;
```

Implementare la seguente funzione C:

```
void matrix_compute_integral(Mat* dest, Mat* src);
```

che dati in input:

- un puntatore ad una matrice dest, preallocata ed avente le stesse dimensioni di src
- un puntatore ad una matrice src

Inserisce in ogni cella  $[r, c]$  della matrice dest la somma degli elementi della sottomatrice di src avente estremi  $[0, 0]$  e  $[r, c]$ .

## Esempio

Data una struttura Mat src contenente la seguente matrice 5x5:

```
3 4 3 3 4 2  
4 4 1 0 -3 3  
3 -4 1 -2 1 -2
```

e data una struttura Mat dest contenente una matrice della stessa dimensione di src, richiamando la seguente linea di codice:

```
matrix_compute_integral(&dest, &src);
```

si ottiene il seguente effetto su dest:

```
3 7 10 13 17 19  
7 15 19 22 23 28  
10 14 19 20 22 25
```

```
#include "esercizio1.h"  
  
void matrix_compute_integral(Mat* dest, Mat* src) {  
    for(int r = 0; r < src->rows; ++r) {  
        for(int c = 0; c < src->cols; ++c) {  
            float sum = 0.0f;  
            for(int rr = 0; rr <= r; ++rr) {  
                for(int cc = 0; cc <= c; ++cc) {  
                    sum += src->row_ptrs[rr][cc];  
                }  
            }  
        }  
    }  
}
```

```
    dest->row_ptrs[r][c] = sum;  
} } }
```

## Esercizio 2

Sia data una lista i cui elementi appartengono alla seguente struttura ListNode:

```
typedef struct ListNode {  
  
    int value;  
  
    struct ListNode* next;  
  
} ListNode;
```

Si realizzi la funzione C:

```
ListNode* find_transitions(int* v, int size)
```

che dato in input un array di interi v di dimensione size, restituisca una lista contenente gli indici i in ordine crescente di tutti gli elementi tali che  $v[i] \neq v[i + 1]$ .

### Esempio

Dato il seguente array di interi v:

1 1 0 0 1 1 0 1

Richiamando la seguente linea di codice:

```
ListNode* l_t = find_transitions(v, 8);
```

l\_t dovrà contenere:

1 -> 3 -> 5 -> 6

```
#include <stdlib.h>  
  
#include "esercizio2.h"  
  
ListNode* find_transitions(int* v, int size) {  
  
    ListNode* gen_node = (ListNode*)calloc(1, sizeof(ListNode));  
  
    gen_node->next = NULL;  
  
    ListNode* list = gen_node;  
  
    for(int i = 0; i < size - 1; ++i) {  
  
        if(v[i] != v[i + 1]) {  
  
            list->next = (ListNode*)calloc(1, sizeof(ListNode));  
            list = list->next;  
        }  
    }  
}
```

```
list = list->next;

list->value = i;

list->next = NULL;

}

}

list = gen_node->next;

free(gen_node);

return list;

}
```

---

## Esercizio 3

Con riferimento alla struttura collegata lineare descritta di seguito:

```
typedef struct ListNode {

    int value;

    struct ListNode* next;

} ListNode;
```

implementare la seguente funzione **ricorsiva** C (nessuna istruzione di ciclo è permessa):

```
int compute_prefix_length(ListNode* l1, ListNode* l2);
```

che, date due liste di interi l1 e l2, restituisca la lunghezza del prefisso comune, ovvero il numero di elementi iniziali uguali.

### Esempio

Date le due seguenti liste di interi l1 e l2:

l1: 5 -> 7 -> 9 -> 1

l2: 5 -> 7 -> 9 -> 11 -> 13 -> 2

Richiamando la seguente linea di codice:

```
int pl = compute_prefix_length(l1, l2);
```

pl dovrà contenere il valore 3

```

#include <stdlib.h>

#include "esercizio3.h"

int compute_prefix_length(ListNode *l1, ListNode *l2) {
    if(l1 == NULL) {
        return 0;
    }

    else if(l2 == NULL) {
        return 0;
    }

    else if(l1->value != l2->value) {
        return 0;
    }

    else {
        return 1 + compute_prefix_length(l1->next, l2->next);
    }
}

```

# Esame di Tecniche di Programmazione del 08/07/2016 - B

## Esercizio 1

Sia data la seguente struttura Mat:

```

typedef struct Mat {
    int cols; // numero di colonne
    int rows; // numero di righe
    float** row_ptrs; // puntatori alle righe
} Mat;

```

Implementare la seguente funzione C:

```
int compute_diagonal_integral(float** v , Mat* src)
```

che, dati in input:

- un puntatore ad un vettore di float v da allocare
- un puntatore ad una matrice quadrata src

allochi il vettore v avente la stessa dimensione della matrica e calcoli le componenti del vettore nel seguente modo: v[k] contiene la somma degli elementi della sottomatrice di src avente estremi [0, 0] e [k, k] inclusi.

## Esempio

Data una struttura Mat src contenente la seguente matrice 3x3:

```
3 4 3  
4 4 1  
3 -4 1
```

la funzione deve restituire il valore 3 e allocare e inserire in v i seguenti valori:

```
3 15 19
```

```
#include "esercizio1.h"  
#include<stdlib.h>  
  
void compute_diagonal_integral(float** v , Mat* src) {  
    if (src->rows!=src->cols)  
        v=0;  
    if (!src->rows)  
        v=0;  
    *v=(float*) malloc(src->rows*sizeof(float));  
    for(int k=0; k<src->rows; k++) {  
        float acc=0;  
        for (int r=0; r<=k; r++)  
            for (int c=0; c<=k; c++)  
                acc+=src->row_ptrs[r][c];  
        (*v)[k]=acc;  
    } }
```

---

## Esercizio 2

Sia data una lista i cui elementi appartengono alla seguente struttura ListNode:

```
typedef struct ListNode {  
    int value;  
    struct ListNode* next;  
} ListNode;
```

Si realizzi la funzione C:

```
ListNode* find_consecutivevalues(int* v, int size)
```

che dato in input un array di interi v di dimensione size, restituisca una lista contenente gli indici i in ordine crescente di tutti gli elementi tali che  $v[i] == v[i + 1]$ .

## Esempio

Dato il seguente array di interi v:

```
0 0 0 1 1 1 0 1 0 0
```

Richiamando la seguente linea di codice:

```
ListNode* l_t = find_transitions(v, 8);
```

l\_t dovrà contenere:

```
0 -> 1 -> 3 -> 4 -> 8
```

```
#include "esercizio2.h"  
  
#include <stdlib.h>  
  
ListNode* find_consecutivevalues(int* v, int size) {  
  
    ListNode* gen_node = (ListNode*)calloc(1, sizeof(ListNode));  
  
    gen_node->next = NULL;  
  
    ListNode* list = gen_node;  
  
    for(int i = 0; i < size - 1; ++i) {  
  
        if(v[i] == v[i + 1]) {  
  
            list->next = (ListNode*)calloc(1, sizeof(ListNode));  
  
            list = list->next;  
  
            list->value = i;  
  
            list->next = NULL;  
  
        } }  
}
```

```
list = gen_node->next;

free(gen_node);

return list;

}
```

---

## Esercizio 3

Con riferimento alla struttura collegata lineare descritta di seguito:

```
typedef struct ListNode {

    int value;

    struct ListNode* next;

} ListNode;
```

implementare la seguente funzione **ricorsiva** C (nessuna istruzione di ciclo è permessa):

```
int compute_prefix_length(ListNode* l1, ListNode* l2);
```

che, date due liste di interi l1 e l2, restituisca la lunghezza del prefisso comune, ovvero il numero di elementi iniziali uguali.

### Esempio

Date le due seguenti liste di interi l1 e l2:

l1: 5 -> 7 -> 9 -> 1

l2: 5 -> 7 -> 9 -> 11 -> 13 -> 2

Richiamando la seguente linea di codice:

```
int pl = compute_prefix_length(l1, l2);
```

pl dovrà contenere il valore 3

```
#include "esercizio3.h"

#include <stdlib.h>

int compute_prefix_length(ListNode *l1, ListNode *l2) {
    if(l1 == NULL) {
        return 0;
    }
    if(l2 == NULL) {
        return 0;
    }
    if(l1->value != l2->value) {
        return 0;
    }
    return 1 + compute_prefix_length(l1->next, l2->next);
}
```

```
}

else if(l2 == NULL) {

    return 0;

}

else if(l1->value != l2->value) {

    return 0;

}

else {

    return 1 + compute_prefix_length(l1->next, l2->next);

} }
```

---

# Esame di Tecniche di Programmazione del 16/09/2016

---

## Esercizio 1

Sia data la seguente struttura Mat:

```
typedef struct Mat {

    int cols; // numero di colonne

    int rows; // numero di righe

    int** row_ptrs; // puntatori alle righe

} Mat;
```

Implementare la seguente funzione C:

```
int matrix_block_sum(Mat* dest, Mat* src, int start_row, int start_col);
```

che dati in input:

- puntatore ad una matrice dest
- puntatore ad una matrice src
- un intero che specifica il numero di riga start\_row
- un intero che specifica il numero di colonna start\_col

sommi gli elementi di dest con quelli di src, nel sottoblocco di dimensione src->rows x src->cols che inizia dalla riga start\_row dalla colonna start\_col. Se l'operazione non e' possibile perche' il sottoblocco in cui effettuare la somma eccede i limiti della matrice di destinazione, la funzione ritorna 0. In caso contrario la funzione ritorna il numero di elementi sommati (ovvero il numero di elementi di src).

## Esempio

Data una struttura Mat dest contenente la seguente immagine 8x8:

```
1 2 3 1 3 3 2 1  
2 3 1 3 3 2 1 0  
1 3 3 2 1 2 1 1  
1 2 3 1 3 3 2 1  
1 3 3 2 1 2 1 1  
1 2 3 1 3 3 2 1  
2 3 1 2 2 1 2 1  
2 3 1 3 3 2 1 0
```

Data una struttura Mat src contenente la seguente matrice 3x3:

```
5 4 7  
4 4 4  
6 4 6
```

Richiamando la seguente linea di codice:

```
matrix_block_sum(dest, src, 2, 2);
```

si ottiene il seguente effetto su dest:

```
1 2 3 1 3 3 2 1  
2 3 1 3 3 2 1 0  
1 3 8 6 8 2 1 1  
1 2 7 5 7 3 2 1  
1 3 9 6 7 2 1 1  
2 3 1 2 2 1 2 1  
1 2 3 1 3 3 2 1  
2 3 1 3 3 2 1 0
```

e la funzione ritorna 9.

Invocando invece:

```
matrix_block_sum(dest, src, 6, 6);
```

dest rimane immutata, e la funzione ritorna 0. L'operazione di copia infatti eccederebbe i limiti della matrice.

```
1 2 3 1 3 3 2 1
```

```
2 3 1 3 3 2 1 0
```

```
1 3 3 2 1 2 1 1
```

```
1 2 3 1 3 3 2 1
```

```
1 3 3 2 1 2 1 1
```

```
2 3 1 2 2 1 2 1
```

```
1 2 3 1 3 3 7 5 ?
```

```
2 3 1 3 3 2 5 4 ?
```

```
???
```

```
#include <stdlib.h>

#include <stdio.h>

#include "esercizio1.h"

int matrix_block_sum(Mat* dest, Mat* src, int start_row, int start_col) {

    if(start_row + src->rows > dest->rows) { return 0; }

    if(start_col + src->cols > dest->cols) { return 0; }

    int r, c;

    for(r = 0; r < src->rows; ++r) {

        for(c = 0; c < src->cols; ++c) {

            dest->row_ptrs[r + start_row][c + start_col] += src->row_ptrs[r][c];

        }

    }

    return src->rows * src->cols;

}
```

## Esercizio 2

Sia data una lista i cui elementi appartengono alla seguente struttura NodoSCL:

```
struct ListNode {  
    int value;  
    struct NodoSCL* next;  
};
```

Si realizzi la funzione

- `ListNode* lowerValues(Mat* m, int value);`

che, dato il puntatore ad una matrice m definita come nell' esercizio precedente, ed un valore intero v, restituisce una lista in cui ciascun elemento contiene il valore di un elemento di m che ha valore strettamente minore di v. Si noti che gli elementi contenuti nella lista devono essere ordinati per numero di riga e per numero di colonna (nell' ordine ottenuto con una scansione della matrice per righe).

### Esempio

Data una struttura Mat m che rappresenta la seguente matrice 2x3:

```
3 1 2  
1 3 4
```

l' esecuzione della istruzione seguente

```
ListNode* l = lowerValues(m, 3);
```

dovrà assegnare alla variabile l il puntatore alla lista:

```
1 -> 2 -> 1
```

```
#include <stdlib.h>  
  
#include "esercizio2.h"  
  
ListNode* lowerValues(Mat* m, int value) {  
    ListNode* gen_node = (ListNode*) malloc(sizeof(ListNode));
```

```

gen_node->next = NULL;

ListNode* current_node = gen_node;

int r, c;

for(r = 0; r < m->rows; ++r) {

    for(c = 0; c < m->cols; ++c) {

        if(m->row_ptrs[r][c] < value) {

            current_node->next = (ListNode*)malloc(sizeof(ListNode));

            current_node = current_node->next;

            current_node->value = m->row_ptrs[r][c];

            current_node->next = NULL;

        } } }

current_node = gen_node->next;

free(gen_node);

return current_node;
}

```

## Esercizio 3

Con riferimento alla struttura collegata lineare descritta di seguito:

```

typedef struct ListNode {

    int value;

    struct ListNode* next;

} ListNode;

```

implementare la seguente funzione **ricorsiva** C (nessuna istruzione di ciclo è permessa):

```
ListNode* compute_derivative(int* v, int size);
```

che dato in input un array di interi v di dimensione size, restituisca una lista in cui l'elemento in posizione i contiene la differenza tra v[i+1] e v[i].

La lista avrà dimensione size - 1 poichè se l'array ha meno di due elementi non e' possibile calcolare la differenza. Nel caso in cui size è minore di 2 la funzione ritorna il valore NULL.

## Esempio

Dato il seguente array di interi v:

7 6 -5 5 5 -4 4 4 -7

Richiamando la seguente linea di codice:

```
ListNode* l_d = compute_derivative(v, 9);
```

l\_d dovrà contenere:

-1 -> -11 -> 10 -> 0 -> -9 -> 8 -> 0 -> -11

```
#include <stdlib.h>

#include "esercizio3.h"

ListNode* compute_derivative_aux(int* v, int size, int index) {

    if(index >= size) {
        return NULL;
    }

    else {
        ListNode *tail = compute_derivative_aux(v, size, index + 1);
        ListNode *new_node = (ListNode*)calloc(1, sizeof(ListNode));
        new_node->value = v[index] - v[index - 1];
        new_node->next = tail;
        return new_node;
    }
}

ListNode* compute_derivative(int* v, int size) {
    if(size < 2) {
        return NULL;
    }

    return compute_derivative_aux(v, size, 1);
}
```

---

# Esame di Tecniche di Programmazione del 20/06/2017

---

## Esercizio 1

Sia data la seguente struttura Mat:

```
typedef struct Mat {  
    int cols; // numero di colonne  
    int rows; // numero di righe  
    int** row_ptrs; // puntatori alle righe  
} Mat;
```

Implementare la seguente funzione C:

```
Mat* pixelizeMatrix(Mat* src);
```

che dato in input:

- un puntatore ad una matrice src

ritorni il puntatore ad una matrice dest (*da allocare*) delle stesse dimensioni di src. In particolare, la cella [r,c] di dest conterra' la somma dei valori di tutte le celle a destra della cella [r,c] se tale somma risulta strettamente maggiore del valore di src[r,c]. Viceversa, conterra' 0.

Si gestisca il caso in cui src sia un puntatore a NULL, ritornando NULL

### Esempio

Data una struttura Mat src contenente la seguente matrice 3x4:

```
9 1 2 7  
0 9 3 6  
0 6 2 6
```

richiamando la seguente linea di codice:

```
Mat* dest = pixelzeMatrix(src);
```

assegni alla variabile dest il puntatore a:

10 9 7 0  
18 0 6 0  
14 8 6 0

```
#include "esercizio1.h"
#include <stdlib.h>

Mat* allocMatrix(int rows, int cols) {
    Mat* matrix = (Mat*)malloc(sizeof(Mat));
    matrix->rows = rows;
    matrix->cols = cols;
    matrix->row_ptrs = (int**)malloc(rows*sizeof(int*));
    for(int r=0; r<rows; ++r)
        matrix->row_ptrs[r] = (int*)malloc(cols*sizeof(int));

    return matrix;
}

Mat* pixelizeMatrix(Mat* src) {
    if(src == NULL)
        return NULL;

    int rows = src->rows;
    int cols = src->cols;

    Mat* result = allocMatrix(rows, cols);
    for(int r = 0; r < rows; ++r){
        for(int c = 0; c < cols; ++c){
            result->row_ptrs[r][c] = 0;
            int valore = src->row_ptrs[r][c];

            int somma = 0;
            for(int i = c+1; i < cols; ++i)
                somma += src->row_ptrs[r][i];

            if(valore < somma)
                result->row_ptrs[r][c] = somma;
        }
    }

    return result;
}
```

## Esercizio 2

Sia data una lista i cui elementi appartengono alla seguente struttura ListElem:

```
typedef struct ListElem {  
    int row;  
    int col;  
    char value;  
    struct ListElem* next;  
} ListElem;
```

contenente nel campo value solo caratteri minuscoli

Sia data inoltre la seguente struttura Mat:

```
typedef struct Mat {  
    int cols; // numero di colonne  
    int rows; // numero di righe  
    char** row_ptrs; // puntatori alle righe (n.b. char)  
} Mat;
```

contenente, nelle celle puntate da row\_ptrs sia valori maiuscoli che minuscoli.

Si realizzi la funzione

- ListElem\* lookupList(Mat\* src\_mat, ListElem\* src\_list);

che dati in input:

- un matrice di caratteri src\_mat
- una lista src\_list di triple(row,col,value)

restituisca una **nuova** lista contenente i soli valori di src\_list tali che:

src\_mat[row][col] == value.

cio' deve valere sia nel caso in cui il carattere presente nella matrice src\_mat sia minuscolo che maiuscolo. Per convertire un carattere da minuscolo a maiuscolo occorre semplicemente sottrarre al carattere minuscolo 32 (es. 'C' == 'c' - 32)

i caratteri con cui popolare la nuova lista saranno sempre minuscoli.

si gestiscano i casi di src\_mat o src\_list vuoti, ritornando una lista vuota.

## Esempio

Data la seguente lista src\_list in ingresso:

[ (1,2) c ] -> [ (1,2) c ] -> [ (0,1) a ] -> [ (1,2) a ] -> [ (0,0) b ] -> [ (1,1) c ]

Data la seguente matrice src\_mat in ingresso

B c a  
c A C

L'esecuzione della seguente istruzione

```
ListElem* dest_list = lookupList(src_mat, src_list)
```

assegni alla variabile dest\_list il puntatore a:

[ (1,2) c ] -> [ (1,2) c ] -> [ (0,0) b ]

```
#include "esercizio2.h"

#include <stdlib.h>

#include <stdio.h>

ListElem* aggiungiElemento(ListElem* l, int row, int col, char value){

    ListElem* new_element=(ListElem*)malloc(sizeof(ListElem));

    new_element->next=NULL;

    new_element->row = row;

    new_element->col = col;

    new_element->value=value;

    if (l==NULL) {

        return new_element;

    }

    ListElem* aux=l;

    while(aux->next!=NULL) {

        aux=aux->next;
```

```
}

aux->next=new_element;

return l;

}

ListElem* lookupList(Mat* src_mat, ListElem* src_list){

if(src_mat == NULL || src_list == NULL)

    return NULL;

ListElem* result_list = NULL;

int rows = src_mat->rows;

int cols = src_mat->cols;

while(src_list != NULL) {

    if(src_list->row <= rows &&

        src_list->col <= cols)

        if(src_mat->row_ptrs[src_list->row][src_list->col] == src_list->value ||

            src_mat->row_ptrs[src_list->row][src_list->col] == src_list->value-32)

            result_list = aggiungiElemento(result_list,

                src_list->row,

                src_list->col,

                src_list->value);

            src_list = src_list->next;

}

return result_list;

}
```

---

## Esercizio 3

Implementare la seguente funzione **ricorsiva** C:

```
void mergeArray(int* src, int* dest, int length);
```

che dati due array src e dest (preallocati) e la loro dimensione length, scriva in dest[i] il minore tra gli elementi *i-esimi* di src e dest.

### Esempio

Dato il seguente array src:

2 6 1 8 7

con length = 4

Dato il seguente array dest:

9 2 0 2 3

con length = 4

La chiamata mergeArray(src, dest, length) assegna alla variabile dest il puntatore a:

2 2 0 2 3

```
#include "esercizio3.h"
#include <stdlib.h>
#include <stdio.h>

void mergeArray(int* src, int * dest, int length) {
    if(length == 1) {
        if(dest[length-1] > src[length-1])
            dest[length-1] = src[length-1];
        return;
    }

    mergeArray(src, dest, length-1);
    if(dest[length-1] > src[length-1])
        dest[length-1] = src[length-1];
}
```

# Esame di Tecniche di Programmazione del 21/07/2017 - A

---

## Esercizio 1

Sia data la seguente struttura Mat:

```
typedef struct Mat {  
    int cols; // numero di colonne  
    int rows; // numero di righe  
    int** row_ptrs; // puntatori alle righe  
} Mat;
```

Implementare la seguente funzione C:

```
Mat* dilateMatrix(Mat* src);
```

che dato in input:

- un puntatore ad una matrice src contenente i soli valori 0 e 1.

ritorni il puntatore ad una **nuova** matrice dest (ovviamente *da allocare*) delle stesse dimensioni di src anch'essa contenente i soli valori 0 e 1. In particolare:

Gli elementi di bordo della matrice dest conterranno 0.

Per quanto riguarda le celle interne alla matrice, [r,c] di dest dovrà contenere il valore 1 se almeno uno degli elementi di un intorno quadrato di src[r,c] ha valore 1, altrimenti dest[r,c] dovrà contenere il valore 0.

Si gestisca il caso in cui src sia un puntatore a NULL, ritornando NULL

## Esempio

Data una struttura Mat src contenente la seguente matrice 5x6:

```
1 0 0 0 0 0  
0 0 0 1 1 0  
0 0 0 0 0 0  
0 0 0 0 0 0  
0 1 1 0 0 0
```

richiamando la seguente linea di codice:

```
Mat* dest = dilateMatrix(src);
```

si ottiene il seguente effetto su dest:

```
0 0 0 0 0  
0 1 1 1 1 0  
0 0 1 1 1 0  
0 1 1 1 0 0  
0 0 0 0 0 0
```

```
#include "esercizio1.h"  
  
#include <stdlib.h>  
  
Mat* allocMatrix(int rows, int cols) {  
    Mat* matrix = (Mat*)malloc(sizeof(Mat));  
    matrix->rows = rows;  
    matrix->cols = cols;  
    matrix->row_ptrs = (int**)malloc(rows*sizeof(int*));  
    for(int r=0; r<rows; ++r)  
        matrix->row_ptrs[r] = (int*)malloc(cols*sizeof(int));  
    return matrix;  
}  
  
Mat* dilateMatrix(Mat* src) {  
  
    if(src == NULL)  
        return NULL;  
    int rows = src->rows;  
    int cols = src->cols;  
  
    Mat* result = allocMatrix(rows, cols);  
  
    for(int r = 0; r < rows; r++) {  
        for(int c = 0; c < cols; c++) {  
            // set borders to zero  
            if(r == 0 || c == 0 || r == rows-1 || c == cols-1) {  
                result->row_ptrs[r][c] = 0;  
                continue;  
            }  
            result->row_ptrs[r][c] = 0;  
            int not_found = 1;  
            for(int rr = r-1; rr <= r+1 && not_found; rr++) {  
                for(int cc = c-1; cc <= c+1; cc++) {  
                    if(src->row_ptrs[rr][cc] != 0) {  
                        result->row_ptrs[r][c] = 1;  
                        not_found = 0;  
                        break;  
                    }  
                }  
            }  
        }  
    }  
    return result;  
}
```

## Esercizio 2

Sia data una lista concatenata lineare i cui elementi appartengono alla seguente struttura ListElem:

```
typedef struct ListElem  
{  
    int value;  
    int occurrences;  
    struct ListElem* next;  
} ListElem;
```

Sia data inoltre la seguente lista concatenata lineare RawListElem:

```
typedef struct RawListElem  
{  
    int value;  
    struct RawListElem* next;  
} RawListElem;
```

Si realizzi la funzione

- `RawListElem * uncompressList ( ListElem *list );`

che dato in input:

- lista concatenata a link singolo list

restituisca il puntatore ad una lista concatenata RawListElem (ovviamente *da allocare*) che contenga una sequenza di interi costruita a partire da list. In pratica la lista list contiene una sequenza di interi "compressa", ove le sequenze identiche di un certo valore sono memorizzate attraverso un solo elemento che contiene il valore (campo value) ed la dimensione della sequenza di quello specifico valore (campo occurrences) nella sequenza originale. La funzione uncompressList() dovrà restituire un una lista che conterrà la sequenza originale, comprese le sequenze di valori identici ripetuti.

Si gestisca il caso in cui il puntatore list e' nullo, restituendo un puntatore nullo.

### Esempio

Data la seguente lista list in ingresso:

[ 8 2 ] -> [ 1 1 ] -> [ 5 3 ]

L'esecuzione della seguente istruzione

```
RawListElem* dest_raw_list = uncompressList(list)
```

produca il seguente effetto su dest\_raw\_list:

[ 8 ] -> [ 8 ] -> [ 1 ] -> [ 5 ] -> [ 5 ] -> [ 5 ]

```
#include "esercizio2.h"

#include <stdlib.h>

#include <stdio.h>

RawListElem* uncompressList(ListElem* src_list) {

    if(src_list == NULL)

        return NULL;

    RawListElem* head = NULL;

    RawListElem* tail = NULL;

    while(src_list !=NULL)  {

        int value = src_list->value;

        int occurrences = src_list->occurrences;

        for(int i = 0; i < occurrences; ++i)  {

            //create new element

            RawListElem* new_elem = (RawListElem*)malloc(sizeof(RawListElem));

            new_elem->value = value;

            new_elem->next = NULL;

            //add element to the list

            if(head == NULL)  {
```

```

head = new_elem;

tail = head;

}

else{

tail->next = new_elem;

tail = tail->next;

} }

src_list = src_list->next;

}

return head;
}

```

## Esercizio 3

Implementare la seguente funzione **ricorsiva** C:

float evaluatePoly( float\* coeff, int length, float x);

che dato array di float coeff di dimensione length che rappresenta la sequenza di coefficienti di un polinomio di grado length - 1:

$$\text{coeff}[0] + \text{coeff}[1] * x + \text{coeff}[2] * x^2 + \dots + \text{coeff}[length-1] * x^{length-1}$$

valuti il polinomio per il valore x passato in input.

### Esempio

Data il seguente array coeff di dimensione 4:

[10 3 0 2]

La chiamata a evaluatePoly( coeff, 4, 2) dovrà restituire il valore 32, ovvero:

$$10 + 3 * 2 + 0 * (2*2) + 2 * (2*2*2) = 32$$

Nota: per valutare la potenza di un valore, e' concesso l'uso della funzione di libreria matematica pow() (header math.h):

double pow (double base, double exponent);

```
#include "esercizio3.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

float evaluatePoly(float* coeff, int length, float x) {
    if(length == 1)
        return coeff[length-1];

    float result = evaluatePoly(coeff,length-1,x);
    return result + coeff[length-1]*pow(x,length-1);
}
```

---

# Esame di Tecniche di Programmazione del 21/07/2017 - B

---

## Esercizio 1

Sia data la seguente struttura Mat:

```
typedef struct Mat
{
    int cols; // numero di colonne
    int rows; // numero di righe
    int** row_ptrs; // puntatori alle righe
} Mat;
```

Implementare la seguente funzione C:

```
Mat* isolateMatrix(Mat* src);
```

che dato in input:

- un puntatore ad una matrice src contenente i soli valori 0 e 1.

ritorni il puntatore ad una matrice dest (ovviamente *da allocare*) delle stesse dimensioni di src anch'essa contenente i soli valori 0 e 1. In particolare:

Le celle sui bordi di dest andranno tutte settate a 0.

Per quanto riguarda gli elementi non di bordo, la cella [r,c] di dest dovrà contenere il valore 1 se src[r,c] ha valore 1 e tutti gli elementi vicini a src[r,c] hanno valore 0, altrimenti dest[r,c] dovrà contenere il valore 0. Si considerano elementi vicini ad una cella src[r,c] tutti gli elementi src[i,j] con  $i = r-1, r, r+1$  e  $j = c-1, c, c+1$  (ad esclusione ovviamente dell'elemento con coordinate  $i=r$  e  $j=c$ ).

Si gestisca il caso in cui src sia un puntatore a NULL, ritornando NULL

## Esempio

Data una struttura Mat src contenente la seguente matrice 5x6:

```
0 0 0 0 0 0  
0 1 0 1 0 1  
0 0 0 0 0 0  
0 0 1 0 1 0  
0 1 1 0 0 1
```

richiamando la seguente linea di codice:

```
Mat* dest = isolateMatrix(src);
```

si ottiene il seguente effetto su dest:

```
0 0 0 0 0 0  
0 1 0 1 0 0  
0 0 0 0 0 0  
0 0 0 0 0 0  
0 0 0 0 0 0
```

```
#include "esercizio1.h"  
#include <stdlib.h>  
  
Mat* matAlloc(int rows, int cols) {  
    Mat* m=(Mat*) malloc(sizeof (Mat));  
    m->cols=cols;  
    m->rows=rows;  
    m->row_ptrs = (int**) malloc(sizeof(int*) * rows);  
    for (int r=0; r<rows; ++r)  
        m->row_ptrs[r] = (int*) malloc(sizeof(int) * cols);  
    return m;  
}  
  
int isolateElement(Mat* src, int row, int col) {  
    for (int r=row-1; r<=row+1; ++r)  
        for (int c=col-1; c<=col+1; ++c) {
```

```

    if (r==row && c==col)
continue;
    if (src->row_ptrs[r][c]==1)
return 0;
}
return 1;
}

void matFill(Mat* src) {
    for (int r=0; r<src->rows; ++r)
        for (int c=0; c<src->cols; ++c)
            src->row_ptrs[r][c]=0;
}

Mat* isolateMatrix(Mat* src) {
    if (! src)
        return NULL;
    Mat* dest=matAlloc(src->rows, src->cols);
    matFill(dest);
    for (int r=1; r<src->rows-1; ++r)
        for (int c=1; c<src->cols-1; ++c) {
            if (src->row_ptrs[r][c]) {
dest->row_ptrs[r][c]=isolateElement(src,r,c);
            }
        }
    return dest;
}

```

## Esercizio 2

Sia data la seguente struttura Vec che rappresenta un vettore (array) di interi:

```

typedef struct
{
    int size; // dimensione del vettore
    int *data; // puntatore agli elementi del vettore
} Vec;

```

Sia data inoltre una lista concatenata lineare i cui elementi appartengono alla seguente struttura ListElem:

```
typedef struct ListElem
```

```
{
```

```
    int value;
```

```
    int occurrences;
```

```
    struct ListElem* next;
```

```
} ListElem;
```

Si realizzi la funzione

- ListElem\* compressVector( Vec \*vec );

che dato in input:

- un vettore di interi vec

restituisca una lista i cui elementi contengano nel campo value i valori incontrati in vec e nel campo occurrences il numero di occorrenze consecutive di value in vec. L'ordine dei valori nella lista dovrà rispettare l'ordine dei valori in vec, per ogni sequenza di un certo valore la lista dovrà contenere un solo nodo, con la dimensione della sequenza settata nel campo occurrences.

Si gestisca il caso in cui vec ha dimensione 0, ritornando una lista vuota.

## Esempio

Data il seguente vettore vec di dimensione 16 in ingresso:

```
[ 8 8 1 5 5 5 4 4 4 4 4 4 4 5 5 3 ]
```

L'esecuzione della seguente istruzione

```
ListElem* dest_list = compressVector(vec)
```

produca il seguente effetto su dest\_list:

```
[ 8 2 ] -> [ 1 1 ] -> [ 5 3 ] -> [ 4 7 ] -> [ 5 2 ] -> [ 3 1 ]
```

```
#include "esercizio2.h"

#include <stdlib.h>

#include <stdio.h>

ListElem* compressVector(Vec* src_vec) {
```

```
int last_value=0;

int sequence_size=0;

ListElem* head=NULL;

for (int i=src_vec->size-1; i>=0; --i) {

    if (last_value!=src_vec->data[i]) {

        if (sequence_size) {

            ListElem * elem=(ListElem*) malloc(sizeof(ListElem));
            elem->value=last_value;
            elem->occurrences=sequence_size;
            elem->next=head;
            head=elem;
        }

        sequence_size=1;
        last_value=src_vec->data[i];
    } else {

        sequence_size++;
        last_value=src_vec->data[i];
    }
}

if (sequence_size) {

    ListElem * elem=(ListElem*) malloc(sizeof(ListElem));
    elem->value=last_value;
    elem->occurrences=sequence_size;
    elem->next=head;
    head=elem;
}

return head;
}
```

# Esercizio 3

Implementare la seguente funzione **ricorsiva** C:

```
float averageVec( float* vec, int length );
```

che dato array di float vec di dimensione length, restituisca la media aritmetica dei valori in esso contenuti.

## Esempio

Data il seguente array vec di dimensione 4:

```
[10 3 0 2]
```

La chiamata ad averageVec( vec, 4 ) dovrà restituire il valore 3.75, ovvero:

$$(10 + 3 + 0 + 2)/4 = 3.75$$

```
#include "esercizio3.h"

#include <stdlib.h>

#include <stdio.h>

float sumVec(int* coeff, int length) {

    if (!length)

        return 0;

    return coeff[0] + sumVec(coeff+1, length-1);

}

float averageVec(int* coeff, int length) {

    if (!length)

        return 0;

    return sumVec(coeff, length)/(float) length;

}
```

# Esame di Tecniche di Programmazione del 12/09/2017

---

## Esercizio 1

Sia data la seguente struttura Mat:

```
typedef struct Mat {  
  
    int cols; // numero di colonne  
  
    int rows; // numero di righe  
  
    int** row_ptrs; // puntatori alle righe  
  
} Mat;
```

Sia inoltre data la struttura MatrixOperation:

```
typedef struct MatrixOperation {  
  
    int row_or_col; // (row_or_col = 0 per scambio RIGHE, row_or_col = 1 per scambio COLONNE)  
  
    int idx1; // indice prima riga o colonna  
  
    int idx2; // indice seconda riga o colonna  
  
} MatrixOperation;
```

Implementare la seguente funzione C:

```
Mat* matrixShuffle(Mat* src, MatrixOperation* ops_mat, int size_ops);
```

che dato in input:

- un puntatore ad una matrice src, un array di MatrixOperation ops\_mat di dimensione size\_ops.

ritorni il puntatore ad una **nuova** matrice dest (ovviamente *da allocare*) delle stesse dimensioni di src i cui valori sono ottenuti dallo scambio di righe e colonne di src, come indicato dagli elementi dell'array ops\_mat.

Si gestisca il caso in cui src sia un puntatore a NULL, ritornando NULL

## Esempio

Data una struttura Mat src contenente la seguente matrice 3x4:

```
6 2 6 1  
8 7 9 2  
0 2 3 7
```

e data la seguente sequenza di operazioni ops\_mat:

```
ops #1: row_or_col=1, idx1=1, idx2=2 // scambia colonna 1 e 2
ops #2: row_or_col=0, idx1=2, idx2=1 // scambia righe 1 e 2
ops #3: row_or_col=1, idx1=1, idx2=0 // scambia colonne 1 e 0
```

richiamando la seguente linea di codice:

```
Mat* dest = matrixShuffle(src, ops_mat, 3);
```

si ottiene il seguente effetto su dest:

```
6 6 2 1
3 0 2 7
9 8 7 2
```

```
#include "eserciziol.h"
#include <stdlib.h>

Mat* allocMatrix(int rows, int cols){
    Mat* mat = (Mat*)malloc(sizeof(Mat));
    mat->rows = rows;
    mat->cols = cols;
    mat->row_ptrs = (int**)malloc(rows*sizeof(int*));
    for(int r = 0; r < rows; ++r)
        mat->row_ptrs[r] = (int*)malloc(cols*sizeof(int));
    return mat;
}

Mat* copyMatrix(Mat* src) {
    if(src == NULL)
        return NULL;
    Mat* result = allocMatrix(src->rows, src->cols);
    for(int r = 0; r < src->rows; ++r)
        for(int c = 0; c < src->cols; ++c)
            result->row_ptrs[r][c] = src->row_ptrs[r][c];
    return result;
}

void shuffleRows(Mat* src, int idx1, int idx2) {
    int rows = src->rows;
    if(idx1 >= rows || idx2 >= rows || idx1 == idx2)
        return;
    int* tmp = src->row_ptrs[idx1];
    src->row_ptrs[idx1] = src->row_ptrs[idx2];
    src->row_ptrs[idx2] = tmp;
}
```

```

src->row_ptrs[idx2] = tmp;
}

void shuffleCols(Mat* src, int idx1, int idx2){
    int cols = src->cols;
    int rows = src->rows;
    if(idx1 >= cols || idx2 >= cols || idx1 == idx2)
        return;
    for(int r = 0; r < rows; ++r){
        int tmp = src->row_ptrs[r][idx1];
        src->row_ptrs[r][idx1] = src->row_ptrs[r][idx2];
        src->row_ptrs[r][idx2] = tmp;
    }
}

Mat* matrixShuffle(Mat* src, MatrixOperation* ops_mat, int size_ops) {
    if(src == NULL)
        return NULL;
    Mat* result = copyMatrix(src);

    if(ops_mat == NULL || size_ops < 1)
        return result;

    for(int i=0; i<size_ops; ++i){
        if(ops_mat[i].row_or_col == 0)
            shuffleRows(result, ops_mat[i].idx1, ops_mat[i].idx2);
        else
            shuffleCols(result, ops_mat[i].idx1, ops_mat[i].idx2);
    }
    return result;
}

```

## Esercizio 2

Sia data una lista i cui elementi appartengono alla seguente struttura IntList:

```

typedef struct IntList
{
    int value;

```

```
struct IntList* next;  
}  
}
```

Si realizzi la funzione

- `IntList * listSelect ( IntList *src, IntList* idx_list );`

che date in input:

- una lista concatenata src, contenente valori interi
- una lista concatenata idx\_list, i cui valori rappresentano indici interi

restituisca il puntatore ad una lista concatenata IntList (ovviamente *da allocare*) che contenga i valori contenuti nella lista src, secondo l'ordine definito dai valori della lista idx\_list.

Si gestisca il caso in cui il puntatore list e' nullo, restituendo un puntatore nullo.

## Esempio

Data la seguente lista src in ingresso:

[ 3 ] -> [ 6 ] -> [ 7 ] -> [ 5 ]

e la lista idx\_list:

[ 1 ] -> [ 3 ]

L'esecuzione della seguente istruzione

```
IntList* dest_raw_list = listSelect(src, idx_list)
```

produca il seguente effetto su dest\_raw\_list:

[ 6 ] -> [ 5 ]

```
#include "esercizio2.h"  
  
#include <stdlib.h>  
  
#include <stdio.h>  
  
IntList* listAt(IntList* list, int pos){  
  
    int i=0;  
  
    while(list != NULL) {  
  
        if(pos == i)  
  
            return list;
```

```
    else

        list = list->next;

        ++i;

    }

    return NULL;
}

IntList* addIntElement(IntList* list, int value){

    IntList* element = (IntList*)malloc(sizeof(IntList));

    element->next = NULL;

    element->value = value;

    if(list == NULL)

        return element;

    IntList* head = list;

    while(list->next != NULL) {

        list = list->next;

    }

    list->next = element;

    return head;
}

IntList* listSelect(IntList* src, IntList* idx_list) {

    IntList* result = NULL;

    while(idx_list != NULL) {

        IntList* element = listAt(src, idx_list->value);

        if(element != NULL) {

            result = addIntElement(result, element->value); }

        idx_list = idx_list->next; }

    return result; }
```

---

## Esercizio 3

Con riferimento alla lista concatenata lineare IntList descritta nel precedente esercizio, implementare la seguente funzione **ricorsiva** C:

```
void list2reverseVector( IntList* coeff, int* dest_array, int length);
```

che data una lista concatenata lineare src\_list riempia nell'ordine inverso un array di interi preallocato dest\_array, di dimensione length.

### Esempio

Data la seguente list src\_list in ingresso:

```
[ 5 ] -> [ 6 ] -> [ 2 ] -> [ 9 ]
```

La chiamata a list2reverseVector( src\_list, dest\_array, 4) avra' il seguente effetto su dest\_array:

```
9 2 6 5
```

```
#include "esercizio3.h"
#include <stdlib.h>
#include <stdio.h>

void list2reverseVector(IntList* src_list, int* dest_array, int length){
    if(src_list == NULL || length < 0)
        return;

    list2reverseVector(src_list->next, dest_array, length-1);
    dest_array[length-1] = src_list->value;

}
```

---

## Esame di Tecniche di Programmazione del 17/01/2018

---

# Esercizio 1

Sia data una stringa memorizzata attraverso una lista concatenata i cui elementi appartengono alla seguente struttura CharList:

```
typedef struct CharList {  
    char c;  
    struct CharList* next;  
} CharList;
```

Implementare la seguente funzione C:

```
int isPalindrome(CharList* list);
```

che dato in input:

- una lista concatenata list non vuota

restituisca 1 se la string rappresentata da list e' palindroma, 0 altrimenti. Si ricordi che una stringa e' palindroma se, letta al contrario, rimane invariate. Ad esempio "anna" e "bob" sono stringhe palindrome, mentre "anno" e "bobo" non sono palindrome.

## Esempio

Data una lista CharList\* lista contenente la seguente string:

```
anna
```

richiamando la seguente linea di codice:

```
int palindrome = isPalindrome(lista);
```

assegni alla variabile palindrome il valore 1

```
#include "esercizio1.h"  
  
#include <stdlib.h>  
  
  
int isPalindrome(CharList* src)  
{  
    int dim = 0;  
    CharList* p = src;
```



## Esercizio 2

Sia data una lista MatList:

```
typedef struct MatList {  
    mat* m;  
    struct MatList* next;  
} MatList;
```

Contenente puntatori alla seguente struttura Mat:

```
typedef struct Mat {  
    int cols; // numero di colonne  
    int rows; // numero di righe  
    int** row_ptrs; // puntatori alle righe  
} Mat;
```

Si realizzi la funzione

- `MatList* matrixRealization( int rows_cols_sum, int init_val );`

che dato in input un intero `rows_cols_sum` che rappresenta la somma del numero di righe e del numero di colonne di una matrice, restituiscia il puntatore ad una lista concatenata `MatList` (ovviamente da allocare) che contenga tutte le possibili matrici in cui la somma del numero di righe e del numero di colonne e' pari a `rows_cols_sum`. Tali matrici dovranno essere inizializzate con un unico valore `init_val` fornito in input. Le matrici dovranno essere memorizzate nella lista per numero di righe crescente, ovvero la prima matrice della lista dovrà avere una riga (`rows_cols_sum-1` colonne), la seconda 2 righe e l'ultima `rows_cols_sum-1` righe e una colonna.  
Nota: ogni matrice in lista andrà allocata singolarmente.

### Esempio

Dati i seguenti valori in ingresso:

`rows_cols_sum = 6`

`init_val = 0`

L'esecuzione della seguente istruzione

```
MatList* dest_list = matrixRealization(rows_cols_sum, init_val)
```

assegni alla variabile `dest_list` il puntatore a:

List Elem 0

0 0 0 0 0

List Elem 1

0 0 0 0

0 0 0 0

List Elem 2

0 0 0

0 0 0

0 0 0

List Elem 3

0 0

0 0

0 0

0 0

List Elem 4

0

0

0

0

0

```
#include "esercizio2.h"
```

```
#include <stdlib.h>
```

```
Mat *createMat( int rows, int cols, int val )
```

```
{
```

```
    Mat *m = (struct Mat *)malloc(sizeof(Mat));
```

```

m->rows = rows;

m->cols = cols;

m->row_ptrs = (int**)malloc(rows*sizeof(int *));

for( int r = 0; r < rows; r++ )

    m->row_ptrs[r] = (int*)malloc(cols*sizeof(int));

for( int r = 0; r < rows; r++ )

    for( int c = 0; c < cols; c++ )

        m->row_ptrs[r][c] = val;

return m;

}

MatList* matrixRealization(int rows_cols_sum, int init_val)

{

MatList *head, *n;

head = n = (MatList*)malloc(sizeof(MatList));



for( int r = 1, c = rows_cols_sum - 1; r < rows_cols_sum; r++, c-- )

{

    Mat *m = createMat(r,c,init_val);

    n->next = (MatList*)malloc(sizeof(MatList));

    n = n->next;

    n->m = m;

    n->next = NULL;

}

n = head;

head = head->next;

free(n);

return head; }

```

# Esercizio 3

Implementare la seguente funzione **ricorsiva** C:

```
void removeChar( char *input_str, char c, char* output_str );
```

che dato in input una stringa input\_str e un carattere c, copia in output\_str la stringa input\_str a meno del carattere c, ovvero rimuovendo tutte le occorrenze di c dalla stringa di input. La funziona dovrà essere case sensitive, ovvero versione maiuscola e versione minuscola di uno stesso carattere non dovranno essere considerate come lo stesso carattere. La stringa output\_str è pre-allocata per contenere, in caso, l'intera stringa di input input\_str.

Ricorda: Una stringa in C è un array di caratteri terminato dal carattere '\0'.

## Esempio

Data la seguente stringa input\_str in ingresso:

"Use gcc to compile your C programs."

La chiamata a removeChar( input\_str, 'C', output\_str ) (con carattere da rimuovere C maiuscolo) dovrà memorizzare in output\_str la seguente stringa:

"Use gcc to compile your programs."

```
#include "esercizio3.h"
#include <stdlib.h>

void removeChar(char* input, char c, char* output)
{
    if( *input == '\0' )
    {
        *output = '\0';
        return;
    }

    if( *input != c )
    {
        *output = *input;
        removeChar(input + 1, c, output + 1);
    }
    else
        removeChar(input + 1, c, output);
}
```

# Esame di Tecniche di Programmazione del 11/07/2018

## Esercizio 1

Sia data la seguente struttura `Mat`:

```
struct Mat {  
  
    int cols; // numero di colonne  
  
    int rows; // numero di righe  
  
    int** row_ptrs; // puntatori alle righe  
  
};
```

Implementare la seguente funzione C:

```
void trasforma( Mat* src, int* vect );
```

che, data in input una matrice `src` e un vettore `vect` (gia' allocato, avente dimensione pari al numero di righe della matrice `src` e contenente valori nulli), assegna a ciascun elemento del vettore la somma degli elementi presenti sulla diagonale discendente verso destra che inizia dall'elemento `src[r,0]`.

### Esempio

Data una struttura `Mat *src` contenente la seguente matrice  $3 \times 4$ :

```
4 2 6 10  
2 9 3 0  
1 3 2 12
```

richiamando la seguente linea di codice:

```
trasforma( src, vect )
```

si ottiene il seguente effetto su `vect`:

```
15 5 1
```

```
#include "esercizio_1.h"  
  
#include <stdio.h>  
#include <stdlib.h>  
  
// SOLUTION
```

```

void trasforma( Mat* src, int *vect ) {
    int rows = src->rows, cols = src->cols;
    for( int r = 0; r < rows; r++ )
    {
        vect[r] = 0;
        for( int i = 0; i < rows-r && i<cols; i++ )
        {
            vect[r] = vect[r]+src->row_ptrs[r+i][i];
        }
    }
}

```

## Esercizio 2

Sia data una lista i cui elementi appartengono alla seguente struttura [ListNode](#):

```

struct ListNode {

    Data info;

    ListNode *next;

};

```

dove ogni elemento contiene un valore intero:

```
typedef int Data;
```

Implementare la seguente funzione C:

```
ListNode *subset(ListNode *list);
```

che data in input una lista concatenata [list](#), restituisca il puntatore ad una nuova lista concatenata che contiene i soli elementi di [list](#) i cui valori sono maggiori dell'elemento ad essi successivo. L'ultimo valore di [list](#) deve essere sempre incluso. L'ordine degli elementi della lista ritornata dovrà essere quello della lista di input.

Si gestisca il caso in cui il puntatore [list](#) è nullo, restituendo un puntatore nullo. La lista in ingresso alla funzione non deve essere modificata in alcun modo.

## Esempio

Data la seguente lista `list` in ingresso:

`39 -> 36 -> 8 -> 35 -> 31 -> 40 -> 30 -> NULL`

L'esecuzione della seguente linea di codice:

```
ListNode* dest_list = subset(list);
```

memorizza in `dest_list` il puntatore a:

`39 -> 36 -> 35 -> 40 -> 30 -> NULL`

```
#include "esercizio_2.h"

#include <stdio.h>

#include <stdlib.h>

// SOLUTION

static ListNode *newNode( Data d ) {

    ListNode *n = (ListNode *)malloc(sizeof(ListNode));

    n->next = NULL;

    n->info = d;

    return n;

}

ListNode *subset( ListNode *list ) {

    ListNode *first, *last;

    first = last = NULL;
```

```
for( ; list != NULL; list = list->next ) {  
    int nextValue = -1;  
  
    if ( list->next )  
  
        nextValue = list->next->info;  
  
    if( (list->info > nextValue) )  
  
    {  
        if ( first == NULL )  
  
            {  
                first = newNode(list->info);  
  
                last = first;  
  
            }  
  
        else  
  
            {  
                last->next = newNode(list->info);  
  
                last = last->next;  
  
            }  
    }  
}  
  
return first;  
}
```

# Esercizio 3

Sia data una lista i cui elementi appartengono alla seguente struttura:

```
struct ListNode {  
    int info;  
    struct ListNode* next;  
};
```

si realizzi la funzione **ricorsiva** C:

```
ListNode* max(ListNode* lista1, ListNode* lista2)
```

Che date in input due liste **lista1** e **lista2** della stessa lunghezza, restituisca una lista della stessa lunghezza,

in cui ogni elemento in posizione **i** contiene il valore massimo tra i corrispondenti elementi in posizione **i** delle liste **lista1** e **lista2**.

## Esempio

Sia **lista1** la seguente sequenza:

```
[-3] -> [-5] -> [4] -> [-10] -> [10] -> [-5] -> NULL
```

Sia **lista2** la seguente sequenza:

```
[6] -> [-5] -> [-5] -> [1] -> [-2] -> [-1] -> NULL
```

Richiamando la seguente istruzione:

```
ListNode* list_max = max(lista1, lista2);
```

Dove **list\_max** dovrà contenere:

```
[6] -> [-5] -> [4] -> [1] -> [10] -> [-1] -> NULL
```

```
#include "esercizio_3.h"  
#include <stdio.h>  
#include <stdlib.h>  
#include <cmath>  
  
// SOLUTION  
ListNode* max(ListNode* lista1, ListNode* lista2) {  
  
    //ds termination criterion  
    if (lista1 == 0 || lista2 == 0) {  
        return 0;  
    }
```

```

//ds create new node (PD)
ListNode* node = (ListNode*)malloc(sizeof(ListNode));

//ds set higher value
if (list1->info > lista2->info) {
    node->info = list1->info;
} else {
    node->info = lista2->info;
}

//ds set next element and return
node->next = max(list1->next, lista2->next);
return node;
}

```

---

# Esame di Tecniche di Programmazione del 10/09/2018

---

## Esercizio 1

Sia data la seguente struttura `Mat`: `struct Mat { int cols; // numero di colonne int rows; // numero di righe int** row_ptrs; // puntatori alle righe };`; Implementare la seguente funzione C: `int massimo(Mat* src)`; che, data in input una matrice `src`, restituisce il valore massimo della colonna la cui somma degli elementi, tra tutte le colonne della matrice, è massima.

### Esempio

Data una struttura `Mat *src` contenente la seguente matrice `3x4`:

```

1 1 1 0
0 2 1 4
0 3 0 0

```

richiamando la seguente linea di codice: `massimo(src)`; si ottiene 3, essendo il massimo valore della seconda colonna, la cui somma degli elementi ( $1+2+3$ ) è massima rispetto a quella delle altre colonne della matrice.

```

#include "esercizio_1.h"

#include <stdio.h>
#include <stdlib.h>

```

```

int massimo(Mat* src) {

    // variables
    int rows = src->rows;
    int cols = src->cols;
    double somma_max      = 0;
    double somma_colonna = 0;
    int colonna_max       = 0;

    // look for maximum column
    for(int c = 0; c < cols; c++) {
        somma_colonna = 0;
        for(int r = 0; r < rows; r++) {
            somma_colonna += src->row_ptrs[r][c];
        }
        if (c == 0 || somma_colonna > somma_max) {
            somma_max = somma_colonna;
            colonna_max = c;
        }
    }

    // compute maximum value of that column
    int max = 0;
    for(int r = 0; r < rows; r++) {
        if (src->row_ptrs[r][colonna_max] > src->row_ptrs[max][colonna_max]) {
            max = r;
        }
    }

    // done
    return src->row_ptrs[max][colonna_max];
}

```

## Esercizio 2

Sia data una lista i cui elementi appartengono alla seguente struttura `ListNode`: `struct ListNode { Data info; ListNode *next; }`; dove ogni elemento contiene un valore intero: `typedef int Data`; Implementare la seguente funzione C: `ListNode *subset(ListNode *list)`; che data in input una lista concatenata `list`, restituisca il puntatore ad una nuova lista concatenata che contiene i soli elementi di `list` i cui valori sono minori dell'elemento ad essi precedente. Il primo valore di `list` deve essere sempre incluso. L'ordine degli elementi della lista ritornata dovrà essere quello della lista di input.

Si gestisca il caso in cui il puntatore `list` è nullo, restituendo un puntatore nullo. La lista in ingresso alla funzione non deve essere modificata in alcun modo.

## Esempio

Data la seguente lista `list` in ingresso:

5 -> 3 -> 8 -> 8 -> 7 -> 0 -> 11 -> NULL

L'esecuzione della seguente linea di codice: `ListNode* dest_list = subset(list);` memorizza in `dest_list` il puntatore a:

5 -> 3 -> 7 -> 0 -> NULL

```
#include "esercizio_2.h"

#include <stdio.h>
#include <stdlib.h>

ListNode *newNode(Data d) {
    ListNode *n = (ListNode *)malloc(sizeof(ListNode));
    n->next = NULL;
    n->info = d;
    return n;
}

ListNode* subset(ListNode *list) {
    ListNode* first = NULL;
    ListNode* last = NULL;
    int precValue = 0;

    // scan complete list
    if (list != NULL) {
        first = newNode(list->info);
        last = first;
        list = list->next;
        precValue = last->info;
    }

    // assemble return list
    for( ; list != NULL; list = list->next ) {
        if( (list->info < precValue) ) {
            last->next = newNode(list->info);
            last = last->next;
        }
        precValue = list->info;
    }

    // done
    return first;
}
```

## Esercizio 3

Sia data una lista i cui elementi appartengono alla seguente struttura `ListNode`: `struct ListNode { int info; struct ListNode* next; }`; Si realizzi la funzione **ricorsiva** C: `ListNode* erase(ListNode* list, int index)`; che, data in input una lista `list`, restituisca una nuova lista `dest_list` in cui manchi solo l'elemento di `list` contenuto nella posizione `index`. Se l'elemento in posizione `index` non esiste, la lista in output dovrà essere uguale a quella in input.

### Esempio

Data la seguente lista `list` in ingresso:

`4 -> 1 -> 3 -> 5 -> NULL`

L'esecuzione della seguente linea di codice: `ListNode* dest_list = erase(list, 0);` memorizza in `dest_list` il puntatore ad una nuova lista che contiene:

`1 -> 3 -> 5 -> NULL`

```
#include "esercizio_3.h"
#include <stdio.h>
#include <stdlib.h>
#include <cmath>

void erase_aux(ListNode* current_node, ListNode* previous_node_, int index) {

    // escape for an invalid call
    if (current_node == NULL) {
        return;
    }

    // if we have the matching element
    if (index == 0) {

        // connect previous with next (if existing)
        if (previous_node_ != NULL) {

            // connect previous with next (if existing)
            if (current_node->next != NULL) {
                previous_node_->next = current_node->next;

                // otherwise finalize list
            } else {
                previous_node_->next = NULL;
            }
        }
    }
}
```

```

    // keep searching
} else {
    erase_aux(current_node->next, current_node, index-1);
}
}

ListNode* erase(ListNode* list, int index) {

    // handle invalid call: invalid index
    if (index < 0 || list == NULL) {
        return list;
    }

    // bookkeep head and check if it needs to be removed
    ListNode* head = list;
    if (index == 0) {
        head = head->next;
    }

    // call helper function
    erase_aux(list->next, list, index-1);

    // done
    return head;
}

```

```

#include <stdio.h>
#include <stdlib.h>

#include "esercizio_1.h"

/*
Esercizio 1

Sia data la seguente struttura Mat:

struct Mat {
    int cols; // numero di colonne
    int rows; // numero di righe
    int** row_ptrs; // puntatori alle righe
};

Implementare la seguente funzione C:

```

```

void prodCont(Mat* src, Mat* dst);

che, date in input una matrice src e una matrice dst (gia' allocata e contenente valori nulli), assegna alla cella [r,c] della matrice di output dst, il valore src[r,c] sommato al prodotto dei 2 elementi a destra e a sinistra di src[r,c]. Se una cella si trova sul bordo, l'elemento mancante vale src[r,c].
```

```

*/
void prodCont( Mat* src, Mat* dst ) {

    int rows = src->rows, cols = src->cols;

    for( int r = 0; r < rows; r++ ) {
        for( int c = 0; c < cols; c++ ) {
            int val = src->row_ptrs[r][c];
            int ll=0, rr=0;
            if (c == 0)
                ll = val;
            else
                ll = src->row_ptrs[r][c-1];
            if (c == cols-1)
                rr = val;
            else
                rr = src->row_ptrs[r][c+1];
            dst->row_ptrs[r][c] = val + ll*rr;
        }
    }
}

```

```

#include <stdio.h>
#include <stdlib.h>

#include "esercizio_2.h"

/*
Esercizio 2

Sia data una lista i cui elementi appartengono alla seguente struttura ListNode:

struct ListNode {

```

```
    Data info;
    ListNode *next;
};
```

dove ogni elemento contiene un valore intero:

```
typedef int Data;
```

Implementare la seguente funzione C:

```
ListNode *subset(ListNode *list, Data m);
```

che data in input una lista concatenata list, restituisca il puntatore ad una nuova lista concatenata che contine i soli elementi di list i cui valori sono maggiori di m. L'ordine degli elementi della lista ritornata dovrà essere quello della lista di input.

Si gestisca il caso in cui il puntatore list è nullo, restituendo un puntatore nullo. La lista in ingresso alla funzione non deve essere modificata in alcun modo.

```
/*
ListNode *addNode(ListNode *l, Data d) {
    ListNode *n = (ListNode *)malloc(sizeof(ListNode));
    n->next = l;
    n->info = d;
    return n;
}
```

```
ListNode *subset(ListNode *list, Data m) {
    if (list==NULL)
        return NULL;
    else if (list->info > m)
        return addNode(subset(list->next, m), list->info);
    else
        return subset(list->next, m);
}
```

---

```
#include <stdio.h>
#include <stdlib.h>

#include "esercizio_3.h"
```

```
/*
```

### Esercizio 3

Sia data la seguente struttura TipoNodoAlbero:

```
struct TipoNodoAlbero {  
    TipoInfoAlbero info;  
    TipoNodoAlbero *sinistro, *destro;  
};
```

dove ogni elemento contiene un valore intero (positivo o negativo):

```
typedef int TipoInfoAlbero;
```

Implementare la seguente funzione C:

```
void scambiaPuntatori(TipoNodoAlbero* nodo);
```

che data la radice di un albero binario completo, modifichi la struttura nel seguente modo:

se il campo info del figlio destro è minore del campo info del figlio sinistro i due figli devono essere scambiati (cioè il figlio sinistro deve diventare figlio destro e viceversa)

altrimenti la struttura deve rimanere invariata

```
*/
```

```
void scambiaPuntatori(TipoNodoAlbero* nodo) {
```

```
    if (nodo==NULL) {  
        return;  
    }  
    else if (nodo->sinistro && nodo->destro) {  
  
        scambiaPuntatori(nodo->sinistro);  
        scambiaPuntatori(nodo->destro);  
  
        if (nodo->sinistro->info > nodo->destro->info) {  
            TipoNodoAlbero* nodo_sinistro_backup = nodo->sinistro;  
            nodo->sinistro = nodo->destro;  
            nodo->destro = nodo_sinistro_backup;  
        }  
    }  
}
```

# Esercitazione 5

## Argomento: Matrici e File

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "es5_mat.h"

Mat* Mat_alloc(int rows, int cols) {
    Mat* m = (Mat*)malloc(sizeof(Mat));
    m->rows = rows;
    m->cols = cols;
    m->row_ptrs = (float**)malloc(rows * sizeof(float*));
    for(int i = 0; i < m->rows; ++i) {
        m->row_ptrs[i] = (float*)malloc(m->cols * sizeof(float));
    }
    return m;
}

Mat* Mat_read(const char *filename) {
    FILE *fp = fopen(filename, "r");
    if(fp == NULL) {
        printf("Errore nell'apertura del file %s... esco!\n", filename);
        fclose(fp);
        return NULL;
    }
    int rows = 0;
    if(fscanf(fp, "%d", &rows) != 1) {
        printf("Errore durante la lettura del file %s, non riesco a leggere il numero di righe... esco!\n", filename);
        fclose(fp);
        return NULL;
    }
    if(rows == 0) {
        return NULL;
    }
    int cols = 0;
    if(fscanf(fp, "%d", &cols) != 1) {
        printf("Errore durante la lettura del file %s, non riesco a leggere il numero di colonne... esco!\n", filename);
        fclose(fp);
        return NULL;
    }
}
```

```

}

if(cols == 0) {
    return NULL;
}

Mat *m = Mat_alloc(rows, cols);
for(int r = 0; r < rows; ++r) {
    for(int c = 0; c < cols; ++c) {
        float valore = 0.0f;
        if(fscanf(fp, "%f", &valore) != 1) {
            printf("Errore durante la lettura del file %s, non riesco a leggere tutti gli elementi... esco!\n", filename);
            fclose(fp);
            return NULL;
        }
        m->row_ptrs[r][c] = valore;
    }
}
fclose(fp);
return m;
}

void Mat_print(Mat *m) {
    for(int r = 0; r < m->rows; ++r) {
        for(int c = 0; c < m->cols; ++c) {
            printf("%lf ", m->row_ptrs[r][c]);
        }
        printf("\n");
    }
}

void Mat_write(const char *filename, Mat *m) {
    FILE *fp = fopen(filename, "w");
    if(fp == NULL) {
        printf("Errore nell'apertura del file %s... esco!\n", filename);
        return;
    }
    fprintf(fp, "%d ", m->rows);
    fprintf(fp, "%d\n", m->cols);
    for(int r = 0; r < m->rows; ++r) {
        for(int c = 0; c < m->cols; ++c) {
            fprintf(fp, "%f ", m->row_ptrs[r][c]);
        }
        fprintf(fp, "\n");
    }
    fclose(fp);
}

```

## Esercizio 5.1

Scrivere la funzione C

```
bool Mat_is_symmetric(Mat *m);
```

che data in ingresso una struttura Mat m contenente una matrice, verifichi che m sia simmetrica o meno. Se m e' simmetrica la funzione deve restituire true in uscita, altrimenti deve restituire false. Si ricorda che una matrice e' simmetrica se ogni elemento  $X_{ij}$  e' uguale all'elemento  $X_{ji}$ .

```
bool Mat_is_symmetric(Mat *m) {
    for(int r = 0; r < m->rows; ++r) {
        for(int c = 0; c < m->cols; ++c) {
            if(m->row_ptrs[r][c] != m->row_ptrs[c][r]) {
                return false;
            }
        }
    }
    return true;
}
```

## Esercizio 5.2

Scrivere la funzione C

```
void Mat_normalize_rows(Mat *m);
```

che, data in ingresso una struttura Mat m contenente una matrice, modifichi m in modo da normalizzare le righe. Si ricorda che la normalizzazione di una riga si ottiene dividendo tutti gli elementi della riga per il modulo della riga stessa.

```
void Mat_normalize_rows(Mat *m) {
    for(int r = 0; r < m->rows; ++r) {
        float squared_sum = 0.0f;
        for(int c = 0; c < m->cols; ++c) {
            squared_sum += m->row_ptrs[r][c] * m->row_ptrs[r][c];
        }
        if(abs(squared_sum) > 1e-3) {
            for(int c = 0; c < m->cols; ++c) {
                m->row_ptrs[r][c] /= sqrtf(squared_sum);
            }
        }
    }
}
```

## Esercizio 5.3

Scrivere la funzione C

```
Mat* Mat_clone(Mat *m);
```

che, data in ingresso una struttura Mat m contenente una matrice, allochi e restituisca una copia della matrice m.

```
Mat* Mat_clone(Mat *m) {
    Mat *copy = Mat_alloc(m->rows, m->cols);

    for(int r = 0; r < copy->rows; ++r) {
        for(int c = 0; c < copy->cols; ++c) {
            copy->row_ptrs[r][c] = m->row_ptrs[r][c];
        }
    }

    return copy;
}

void Mat_free(Mat *m) {
    for(int i = 0; i < m->rows; ++i) {
        free(m->row_ptrs[i]);
    }
    free(m->row_ptrs);
    free(m);
}
```

## Esercizio 5.4

Scrivere la funzione C

```
Mat* Mat_sum(Mat *m1, Mat *m2);
```

che, date in ingresso due strutture Mat m1 e Mat m2 contenenti due matrici, allochi e restituisca la somma delle suddette matrici. Nel caso non fosse possibile eseguire la somma (per esempio, se le dimensioni delle due matrici di input non sono uguali), la funzione deve stampare a schermo un messaggio di errore e ritornare NULL.

```
Mat* Mat_sum(Mat *m1, Mat *m2){

    if( m1->rows != m2->rows || m1->cols != m2->cols){
        printf("Operazione di somma non consensita!\n");
        return NULL;
    }

    Mat *output = Mat_alloc(m1->rows, m1->cols);
    for(int r = 0; r < output->rows; ++r) {
```

```

        for(int c = 0; c < output->cols; ++c) {
            output->row_ptrs[r][c] = m1->row_ptrs[r][c] + m2->row_ptrs[r][c];
        }
    }

    return output;
}

```

## Esercizio 5.5

Scrivere la funzione C

`Mat* Mat_product(Mat *m1, Mat *m2);`

che, date in ingresso due strutture Mat m1 e Mat m2 contenenti due matrici, allochi e restituisca il prodotto delle suddette matrici. Nel caso non fosse possibile eseguire il prodotto (per esempio, se le dimensioni delle due matrici di input non consentono il prodotto), la funzione deve stampare a schermo un messaggio di errore e ritornare NULL.

```

Mat* Mat_product(Mat *m1, Mat *m2){

    if( m1->cols != m2->rows){
        printf("Operazione di prodotto matriciale non consensita!\n");
        return NULL;
    }

    Mat *output = Mat_alloc(m1->rows, m2->cols);

    for(int r = 0; r < output->rows; ++r) {
        for(int c = 0; c < output->cols; ++c) {
            output->row_ptrs[r][c] = 0.f;
            for(int k=0; k<output->rows; k++){
                output->row_ptrs[r][c] += m1->row_ptrs[r][k] * m2->row_ptrs[k][c];
            }
        }
    }

    return output;
}

```

## Esercizio 5.6

Scrivere la funzione C che, date in ingresso una struttura Mat m1, allochi e restituisca una struttura Mat m2 contenente tutti elementi con valore 0 o 1. Gli elementi di m2 posti a 1 sono quelli che nella matrice m1 hanno valore massimo tra quelli del loro intorno (di 8 caselle per gli elementi interni, meno per gli elementi sul bordo).

```
Mat* max_intorno(Mat *m1){
    Mat *m2 = Mat_alloc(m1->rows, m1->cols);
    int i, j, i2, j2, max; float v;

    for(i = 0; i < m2->rows; i++)
        for(j = 0; j < m2->cols; j++)
            m2->row_ptrs[i][j] = 0;

    for(i = 0; i < m1->rows; i++)
        for(j = 0; j < m1->cols; j++)
    {
        v = m1->row_ptrs[i][j];
        max = 1;
        for(i2 = i - 1; i2 < i + 2; i2++)
            for(j2 = j; j2 < j + 2; j2++)
                if(i2 >= 0 && i2 < m1->rows && j2 >= 0 && j2 < m1->cols)
                    max = max && (v >= m1->row_ptrs[i2][j2]);
        if(max)
            m2->row_ptrs[i][j] = 1;
    }

    return m2;
}
```

Esempio:

Input: m1	Output: m2
[[1 2 3 3]	[[0 0 0 0]
[2 13 7 8]	[0 0 0 1]
[9 20 6 7]]	[0 1 0 0]

# Esercitazione 6

## Argomento: Ricorsione su Stringhe e Array

Scaricare i file associati all'esercitazione e completare le definizioni delle funzioni presenti nel file **e6\_esercizio.c**. Compilare e testare il tutto tramite il programma main di prova fornito che si trova all'interno di **e6\_test\_esercizio.c**.

**Nota:** tutti gli esercizi vanno svolti in maniera ricorsiva

```
#include "e6_correttore.h"

void vec_print(float v[], int dim) {
    printf("[");
    for(int i = 0; i < dim - 1; ++i) {
        printf("%f ", v[i]);
    }
    printf("%f]\n", v[dim - 1]);
}
```

### Esercizio 6.1

Scrivere la funzione C

```
int lunghezza(char* s);
```

che, data in ingresso una stringa, calcoli e restituisca la lunghezza della stringa.

```
int _lunghezza(char *s){

    if (*s=='\0'){
        return 0;
    }
    else{
        return 1 + _lunghezza(s+1);
    }
}
```

### Esercizio 6.2

Scrivere la funzione C

```
int char_in_posizione(char* s, char ch);
```

che, data in ingresso una stringa e un carattere, calcoli e restituisca la posizione del primo carattere **ch** nella stringa. Se il carattere non e' presente, la funzione deve restituire -1.

```

int _char_in_posizione(char *s, char ch){

    int i = 0;
    _char_in_posizione_aux(s, ch, i);
}

int _char_in_posizione_aux(char *s, char ch, int i){

    if (*s == '\0'){
        return -1;
    }
    else if (*s == ch){
        return i;
    }
    else{
        return _char_in_posizione_aux(s+1, ch, i+1);
    }
}

```

## Esercizio 6.3

Scrivere la funzione C

```
void vec_integral(float* v, int n);
```

che scrive nell'elemento dell'array `v` con indice `i` la somma di tutti gli elementi che lo precedono (da `0` a `i-1`).

```

void _vec_integral(float* v, int n){
    if(n==0)
        return;

    int i = 0;
    _vec_integral_aux(v, n, i, 0);
}

```

## Esercizio 6.4

Implementare funzione C

```
float vec_dot(float* v1, float* v2, int n);
```

che calcola e restituisce il prodotto scalare tra i vettori  $v_1$  e  $v_2$  (entrambi di dimensione  $n$ ). Es:

[ 4 ]

[1 3 -2] [-2] = 0

[-1]

```
void _vec_integral_aux(float* v, int n, int i, float sum_till_now){  
    if ( i == n)  
        return;  
    *v += sum_till_now;  
    i++;  
    _vec_integral_aux(v+1, n, i, *v);  
}  
  
float _vec_dot(float* v1, float* v2, int n) {  
  
    int i = 0;  
    _vec_dot_aux(v1, v2, n, i);  
}  
  
float _vec_dot_aux(float* v1, float* v2, int n, int i) {  
    if(i == n)  
        return 0;  
    return *v1 * *v2 + _vec_dot_aux(v1+1, v2+1, n, i+1);  
}
```

# Esercitazione 7

## Argomento: Ricorsione su vettori, stringhe e SCL

```
#pragma once
#include "e7_correttore.h"

void vec_print(float v[], int dim) {
    printf("[");
    for(int i = 0; i < dim - 1; ++i) {
        printf("%f ", v[i]);
    }
    printf("%f]\n", v[dim - 1]);
}
```

### Esercizio 7.1

Scrivere la funzione C

```
bool tuttiMinuscoli(char* s)
```

che, data in ingresso una stringa, restituisce 1 se la stringa contiene solo caratteri minuscoli, 0 altrimenti.

```
bool _tuttiMinuscoli(char *s){

    if (*s == '\0'){
        return true;
    }
    else if (*s >= 'a' && *s <= 'z'){
        return _tuttiMinuscoli(s + 1);
    }
    else{
        return false;
    }
}
```

### Esercizio 7.2

Scrivere la funzione C

```
void converti(char* s);
```

che converte tutti i caratteri minuscoli presenti nella stringa di input s nei corrispondenti caratteri maiuscoli.

```
void _converti(char *s){
    if (*s == '\0'){

    }
```

```

        return;
    }
    else if (*s >= 'A' && *s <= 'Z'){
        *s = *s - 'A' + 'a';
        return _converti(s + 1);
    }
    else{
        return _converti(s + 1);
    }
}

```

## Esercizio 7.3

Scrivere la funzione C

```
int contaParentesi(char *s);
```

che, data in ingresso una stringa, calcoli e restituisca il numero di parentesi presenti nella stringa. Considerare come parentesi i caratteri: ([{}]).

```

int _contaParentesi(char *s){

    if (*s == '\0'){
        return 0;
    }
    else if (*s == '(' || *s == '[' || *s == '{' || *s == ')' || *s == ']' || *s ==
')'){
        return 1 + _contaParentesi(s+1);
    }
    else{
        return _contaParentesi(s+1);
    }
}

```

## Esercizio 7.4

Scrivere la funzione C

```
void concatenate(char* dest, char* src);
```

che, date in ingresso due stringhe, concatenai la stringa `src` a `dest` e la memorizzi in `dest`.

```
void _concatenate(char* dest, char* src){  
  
    if (*dest == '\0') {  
        _copia(dest,src);  
    }  
    else {  
        return _concatenate(dest+1, src);  
    }  
}
```

## Esercizio 7.5

Scrivere la funzione C

```
int prodotto(int a[], int n);
```

che, dati in ingresso un array `a` e la sua lunghezza `n`, calcoli e restituisca il prodotto degli elementi di `a`. (la ricorsione va eseguita sia su `a` ed `n`).

```
void _copia(char* dest, char* s){  
  
    if (*s == '\0'){  
        *dest = '\0';  
        return;  
    }  
    else{  
        *dest = *s;  
        return _copia(dest + 1, s + 1);  
    }  
}  
float _prodotto(float a[], int n){  
  
    if (n == 0){  
        return 1;  
    }  
    else{  
        return a[0] * _prodotto(a + 1, n - 1);  
    }  
}
```

## Esercizio 7.6

Sia data la seguente struttura collegata `TipoSCL`:

```
typedef float TipoInfoSCL;
struct Elemscl {
    TipoInfoSCL info ;
    struct Elemscl* next;
};

typedef struct Elemscl TipoNodoSCL;
typedef TipoNodoSCL * TipoSCL;
```

Implementare la funzione:

```
float SCL_media(TipoSCL head_ptr);
```

che restituisce la media degli elementi della lista, il cui puntatore di testa e' `head_ptr`. Eseguire l'esercizio implementando due funzioni ausiliare ricorsive: una che calcola la lunghezza della lista e l'altra che calcola la somma dei suoi elementi.

```
float _SCL_media_aux1(TipoSCL head_ptr){
    if(head_ptr == NULL )
        return 0.0;
    return (head_ptr->info) + _SCL_media_aux1(head_ptr->next);
}

int _SCL_media_aux2(TipoSCL head_ptr){
    if(head_ptr == NULL )
        return 0;
    return 1 + _SCL_media_aux2(head_ptr->next);
}

float _SCL_media(TipoSCL head_ptr){

    return _SCL_media_aux1(head_ptr) / (float) _SCL_media_aux2(head_ptr);
}
```

## Esercizio 7.7

Implementare la funzione:

```
void SCL_integral(TipoSCL head_ptr);
```

che scrive, in ogni elemento della lista (il cui puntatore di testa e' `head_ptr`) la somma degli elementi precedenti (dal primo a se stesso).

```
int _SCL_integral_aux(TipoSCL ptr, float sum){
    if(ptr == NULL){
        return 0;
    }
    ptr->info = ptr->info + sum;
    return _SCL_integral_aux(ptr->next, ptr->info);
}

void _SCL_integral(TipoNodoSCL* head_ptr){
    _SCL_integral_aux(head_ptr, 0);
}
```

## Esercizio 7.8

Implementare la funzione:

```
float SCL_dot(TipoSCL head1_ptr, TipoSCL head2_ptr);
```

che ritorna il prodotto scalare, delle liste con puntatori alla testa `head1_ptr` e `head2_ptr`.

```
float _SCL_dot(TipoSCL head1_ptr, TipoSCL head2_ptr){

    if(head1_ptr == NULL)
        return 0;
    return head1_ptr->info * head2_ptr->info + _SCL_dot(head1_ptr->next, head2_ptr->
next);
}
```

# Esercitazione 8

## Argomento: SCL

Per questa esercitazione l'organizzazione del codice nei vari file e il testing delle funzioni è lasciata agli studenti.

Scaricate la cartella es8 e completate i file forniti. In particolare:

- l'implementazione delle funzioni va scritta nel file functions.c
- le intestazioni delle funzioni va messa nel file functions.h
- il main è nel file test.c
- I file scl.c e scl.h contengono implementazione e intestazioni di funzioni utili per lavorare con le scl, che potete usare senza doverle re-implementare (es creazione scl random, stampa scl etc).

Compilate con l'istruzione

**make**

Eseguite l'eseguibile 'test'

**./test**

## Esercizio 8.1

Sia data la seguente struttura collegata **TipoSCL**:

```
typedef float TipoInfoSCL;
struct Elemscl {
    TipoInfoSCL info ;
    struct Elemscl* next;
};
typedef struct Elemscl TipoNodoSCL;
typedef TipoNodoSCL * TipoSCL;
```

Implementare la funzione:

```
TipoSCL doubledCopy( TipoSCL list );
```

che data una SCL di input, allochi e restituisca il puntatore ad una nuova SCL che contiene, per ogni elemento della SCL di input, due ripetizioni contigue dello stesso. L'ordine degli elementi ripetuti dovrà essere quello della SCL di input.

Si gestisca il caso in cui la SCL di input è nulla, restituendo un puntatore nullo. La SCL in ingresso alla funzione non deve essere modificata in alcun modo.

```
#pragma once

#include "scl.h"
#include "functions.h"
#include<stdlib.h>
#include<stdio.h>
#include<math.h>

TipoSCL doubledCopy( TipoSCL list )
{
    if( list==NULL)
        return NULL;
    else
    {
        TipoSCL node1 = newNode(list->info), node2 = newNode( list->info);

        node2->next = doubledCopy(list->next);
        node1->next = node2;

        return node1;
    }
}
```

## Esercizio 8.2

Implementare la funzione:

```
TipoSCL multipleSubset( TipoSCL list, unsigned int m );
```

che data in input una SCL list, restituisca il puntatore ad una nuova SCL che contiene i soli elementi di list i cui valori sono multipli di m (si usi la convenzione in cui lo 0 è multiplo di tutti i numeri interi). L'ordine degli elementi della SCL restituita dovrà essere quello della SCL di input.

Si gestisca il caso in cui il puntatore list è nullo, restituendo un puntatore nullo. La lista in ingresso alla funzione non deve essere modificata in alcun modo.

Nota: Il valore dell'info è di tipo float nella struttura SCL fornita, perciò l'info va prima trasformata in intero e poi ne va valutato il valore assoluto, in modo da ottenere un intero positivo.

```
TipoSCL multipleSubset( TipoSCL list, unsigned int m )
{
    if( list==NULL )
        return NULL;

    else if( abs( (int) (list->info) ) % m )
    {
        return multipleSubset( list->next, m );
    }
    else
    {
        TipoSCL node = newNode( list->info );
        node->next = multipleSubset( list->next, m );
        return node;
    }
}
```

## Esercizio 8.3

Implementare la seguente funzione C:

```
TipoSCL pairSubset( TipoSCL list );
```

che data in input una SCL list, restituisca il puntatore ad una nuova SCL che contiene i soli elementi di list il cui indice è pari, assumendo che il primo elemento abbia indice 0. L'ordine degli elementi della SCL di output dovrà essere quello della SCL di input.

Si gestisca il caso in cui il puntatore list è nullo, restituendo un puntatore nullo. La SCL in ingresso alla funzione non deve essere modificata in alcun modo.

```
TipoSCL pairSubset( TipoSCL list, int i )
```

```

{
    if( list==NULL )
        return NULL;
    else if( i%2 )
    {
        return pairSubset( list->next, i+1 );
    }
    else
    {
        TipoSCL node = newNode( list->info );
        node->next = pairSubset( list->next, i+1 );
        return node;
    }
}

```

## Esercizio 8.4

Implementare la seguente funzione C:

```
TipoSCL getPrev(TipoSCL list, TipoSCL n);
```

che data in input una SCL list e il puntatore n a un elemento della lista, restituisca il puntatore all'elemento precedente di n.

```

TipoSCL getPrev (TipoSCL h, TipoSCL n)
{
    if (n == NULL || h->next == NULL || h == NULL)
    {
        return NULL;
    }
    else if (h->next == n)
    {
        return h;
    }
    else
    {
        return getPrev (h->next, n);
    }
}

```

## Esercizio 8.5

Implementare la seguente funzione C:

```
TipoSCL getTail(TipoSCL list);
```

che data in input una SCL list, restituisca il puntatore all'ultimo elemento della lista.

```
TipoSCL getTail (TipoSCL h)
{
    if (h->next == NULL)
    {
        return h;
    }
    else
    {
        return getTail (h->next);
    }
}
```

# Esercitazione 10

Argomento: Tipi di dato astratti

Per questa esercitazione si consiglia di organizzare il codice come segue:

- fare un file .h dove mettere le intestazioni delle funzioni
- fare un file .c contenente l'implementazione delle funzioni
- scrivere il main in un altro file .c in cui si testano le funzioni implementate

## Code

### Esercizio 10.1

Implementare una versione con side effect senza condivisione di memoria del tipo di dati astratto **Coda**, rappresentato tramite una SCL, che contenga i puntatori al primo e all'ultimo elemento della coda.

```
typedef int T ;  
  
struct NodoSCL {  
    T info ;  
    struct NodoSCL * next ;  
};  
  
typedef struct NodoSCL TipoNodo ;
```

```
struct CodaColl {  
    TipoNodo * head;  
    TipoNodo * tail;  
};  
  
typedef struct CodaColl CodaColl;  
  
typedef CodaColl * Coda;
```

Implementare perciò le seguenti funzioni:

- Coda \* codaVuota ()
- bool estVuota ( Coda \* c )

- void inCoda ( Coda \* c , T e )
- void outCoda ( Coda \* c )
- T primo ( Coda \* c )
- T ultimo ( Coda \* c )

Ogni funzione va implementata con costo  $O(1)$ .

```
#include "coda.h"
#include<stdlib.h>
#include<stdio.h>

Coda *codaVuota(){
    Coda *nc = (Coda *)malloc(sizeof(Coda));
    Coda c = (Coda)(malloc(sizeof(CodaColl)));
    *nc = c;
    (*nc)->head = NULL;
    (*nc)->tail = NULL;
    return nc;
}

bool estVuota ( Coda * c )
{
    return ((*c)->head == NULL && (*c)->tail == NULL ) ;
}

void inCoda ( Coda * c , T e ){
    TipoNodo * n_nodo = (TipoNodo*)(malloc(sizeof(TipoNodo)));
    n_nodo->info = e;
    n_nodo->next = NULL;
    if(estVuota(c)){
        (*c)->head = n_nodo;
        (*c)->tail = n_nodo;
        return;
    }else{
        (*c)->tail->next = n_nodo;
        (*c)->tail = (*c)->tail->next;
    }
}

void outCoda ( Coda * c )
{
    if ( c == NULL || estVuota(c) ) {
```

```

    printf ( " ERRORE : input NULL o coda vuota " ) ;
    exit (1) ;
}
TipoNodo *tmp = (*c)->head ;
(*c)->head = ((*c)->head )->next ;
free ( tmp ) ;
}

T primo ( Coda * c )
{
    if (estVuota(c) ) {
        printf ( " ERRORE : coda vuota " ) ;
        exit (1) ;
    }
    return (*c)->head->info ;
}

T ultimo ( Coda * c )
{
    if ( estVuota(c) ) {
        printf ( " ERRORE : coda vuota " ) ;
        exit (1) ;
    }
    return (*c ) -> tail->info ;
}

int main(){

    Coda *ptr = codaVuota();
    printf("%d\n",estVuota(ptr));
    inCoda(ptr, 5);
    printf("%d\n",estVuota(ptr));
    printf("%d\t%d\n",primo(ptr), ultimo(ptr));
    inCoda(ptr,4);
    printf("%d\t%d\n",primo(ptr), ultimo(ptr));
    outCoda(ptr);
    printf("%d\t%d\n",primo(ptr), ultimo(ptr));

}

```

# Liste

Si consideri il tipo di dati astratti lista rappresentato mediante una SCL.

```
typedef int T ;  
  
struct NodoSCL {  
  
    T info ;  
  
    struct NodoSCL * next ;  
  
};  
  
typedef struct NodoSCL TipoNodo ;  
  
typedef TipoNodo * TipoLista ;
```

Vi vengono forniti i file lista.c e lista.h, contenenti l' implementazione del tipo secondo uno schema funzionale con condivisione di memoria, e le funzioni che avete visto a lezione su questo tipo di dato (*length*, *append*, *concat*, *ins*, *cons* e *get*).

Utilizzare le funzioni fornite per svolgere gli esercizi seguenti senza accedere direttamente alla SCL che rappresenta la lista:

## Esercizio 10.2

Implementare la funzione C

**T sommaElementi(TipoLista l);**

che data in ingresso una lista la restituisca la somma dei suoi elementi, oppure 0 se la lista è vuota.

## Esercizio 10.3

Implementare la funzione C

**TipoLista halfMerge(TipoLista l1, TipoLista l2);**

che date in ingresso due liste l1 e l2 restituisca una nuova lista formata dalla primà metà della prima lista concatenata con la seconda metà della seconda lista.

Es: input: 2→3→7→18→2→4→29→3→17

output: 2→3→7→3→17

## Esercizio 10.4

Implementare la funzione ricorsiva C

**TipoLista appendNtimes ( TipoLista l , T e, int n);**

che data in ingresso una lista l, un elemento e un intero positivo n, restituisca una lista a cui è stato aggiunto n volte l'elemento e in fondo alla lista.

```
#include "lista.h"

#include<stdio.h>
#include<stdlib.h>

TipoLista listaVuota () { return NULL ;}

int estVuota ( TipoLista l ) { return ( l == NULL ) ;}

TipoLista cons ( T e , TipoLista l ) {
    TipoLista nuovo = ( TipoLista ) malloc ( sizeof ( TipoNodo));
    nuovo -> info = e ;
    nuovo -> next = l ;
    return nuovo ;
}

T car ( TipoLista l ) {
    if ( l == NULL ) {
        printf ( " ERRORE : lista vuota \n " ) ;
        exit (1) ;
    }
    return l -> info ;
}

TipoLista cdr ( TipoLista l ) {
    if ( l == NULL ) {
        printf ( " ERRORE : lista vuota \n " ) ;
        exit (1) ;
    }
    return l -> next ;
}

int length ( TipoLista l ) {
    if ( estVuota ( l ) ) return 0;
    return 1 + length ( cdr ( l ) ) ;
```

```

}

TipoLista append ( TipoLista l , T e ) {
    if ( estVuota ( l ) ) {
        return cons ( e , l ) ;
    }
    return cons ( car ( l ) , append ( cdr ( l ) ,e ) ) ;
}

TipoLista concat ( TipoLista l1 , TipoLista l2 ) {
    if ( estVuota ( l2 ) ) {
        return ( l1 ) ;
    }
    return ( concat ( append ( l1 , car ( l2 ) ) , cdr ( l2 ) ) ) ;
}

T get ( TipoLista l , int i ) {
    if ( i < 0 || estVuota ( l ) ) {
        printf ( " ERRORE : lista vuota o indice fuori dai limiti !\n " ) ;
        exit (1) ;
    }
    if ( i ==0) return car ( l ) ;
    return get ( cdr ( l ) ,i -1) ;
}

TipoLista ins ( TipoLista l , int i , T e ) {
    if ( i < 0 || (i >0 && estVuota ( l ) ) ) {
        printf ( " ERRORE : indice fuori dai limiti !\n " ) ;
        exit (1) ;
    }
    if ( i ==0) return cons ( e , l ) ;
    return ( cons ( car ( l ) , ins ( cdr ( l ) , i -1 , e ) ) ) ;
}

void printLista(TipoLista l1){
    while(!estVuota(l1)){
        printf("[%d]->", car(l1));
        l1 = cdr(l1);
    }
    printf("\n");
    return;
}

int sum ( TipoLista l ){
    if ( estVuota ( l ) ) return 0;
}

```

```

        return car(l) + sum ( cdr (l));
    }

TipoLista appendNtimes ( TipoLista l , T e, int n){
    if( estVuota (l)){
        if(n > 1){
            return cons( e,appendNtimes (l,e,n-1) );
        }else{
            return cons (e,l) ;
        }
    }
    return cons( car(l),appendNtimes (cdr(l),e,n) );
}

TipoLista merge ( TipoLista l1 , TipoLista l2 ){
    if ( estVuota(l2) && !estVuota(l1)){
        return l1;
    }else if( !estVuota(l2) && estVuota(l1)){
        return l2;
    }else if(estVuota (l2) && estVuota (l1)){
        return listaVuota();
    }
    int start1 = length(l1)/2;
    int start2 = length(l2)/2;
    TipoLista n = listaVuota();
    TipoLista tmp = l1;
    for(int i = 0; i < start1; i++){
        n = append(n,car(tmp));
        tmp = cdr(tmp);
    }
    tmp = l2;
    int i = 0;
    while(i < start2){
        tmp = cdr(tmp);
        i++;
    }
    n = concat(n,tmp);
    return n;
}

int main(){

    TipoLista l1 = listaVuota();
    TipoLista l2 = listaVuota();
}

```

```
l1 = append(l1, 1);
l1 = append(l1, 2);
l1 = append(l1, 3);
l1 = appendNtimes(l1,4,6);
l1 = append(l1, 5);
l2 = append(l2, 6);
l2 = append(l2, 7);
l2 = append(l2, 8);
l2 = append(l2, 9);
printLista(l1);
printLista(l2);
l1 = merge(l1,l2);
printLista(l1);
printf("%d\n",sum(l1));
}
```

# Esercitazione 11

## Argomento: alberi binari

Per questa esercitazione si consiglia di organizzare il codice come segue:

- fare un file .h dove mettere le intestazioni delle funzioni
- fare un file .c contenente l'implementazione delle funzioni
- scrivere il main in un altro file .c in cui si testano le funzioni implementate

Vengono forniti i file **alberoBinario.h** e **alberoBinario.c**, contenenti la definizione della struttura dati da utilizzare e alcune funzioni di utilità già viste a lezione, che non dovete re-implementare.

### Esercizio 11.1

Implementare la funzione C

```
int DimensioneAlbero(TipoAlbero a);
```

che, dato un albero binario, restituisca il numero di nodi contenuti nell'albero.

```
#include "correttore_esercizio.h"

int DimensioneAlbero_soluzione(TipoAlbero a){

    if (a == NULL)
        return 0;

    return DimensioneAlbero_soluzione(a->sinistro) + DimensioneAlbero_soluzione(a->destro)+1;
}
```

### Esercizio 11.2

Implementare la funzione

```
TipoInfoAlbero TrovaMassimo(TipoAlbero a);
```

che, dato un albero binario, restituisca il valore massimo contenuto nei nodi dell'albero.

```
TipoInfoAlbero TrovaMassimo_soluzione(TipoAlbero a){
```

```

if (a == NULL)
    return ERRORE_InfoAlbero;

int max = a->info;
int max_sx = TrovaMassimo_soluzione(a->sinistro);
if (max_sx > max)
    max = max_sx;

int max_dx = TrovaMassimo_soluzione(a->destro);
if (max_dx > max)
    max = max_dx;

return max;
}

```

## Esercizio 11.3

Implementare la funzione

```
int contaFraMinMax(TipoAlbero a, TipoInfoAlbero min, TipoInfoAlbero max);
```

che, dati un albero binario e due valori TipoInfoAlbero min e max, restituisca il numero di nodi contenuti nell'albero i cui valori sono compresi nell'intervallo  $\text{]min, max[}$ , con min e max esclusi.

```

int contaFraMinMax_soluzione(TipoAlbero a, const TipoInfoAlbero min, const TipoInfoAlbero max){
    if (a == NULL)
        return 0;

    if (a->info < max && a->info > min) {
        return contaFraMinMax_soluzione(a->sinistro, min, max) + contaFraMinMax_soluzione(a->destro, min, max)+1;
    }

    return contaFraMinMax_soluzione(a->sinistro, min, max) + contaFraMinMax_soluzione(a->destro, min, max);
}

```

## Esercizio 11.4

Implementare la funzione

```
TipoInfoAlbero sommaValoriFoglie(TipoAlbero a);
```

che, dato un albero binario, ritorni la somma di tutti i valori contenuti nei nodi foglia.

```
TipoInfoAlbero sommaValoriFoglie_soluzione(TipoAlbero a){  
  
    if( a == NULL)  
        return 0;  
  
    if( a->sinistro == NULL && a->destro == NULL )  
        return a->info;  
  
    return sommaValoriFoglie_soluzione(a->sinistro) + sommaValoriFoglie_soluzione(  
a->destro);  
}
```

# Esercitazione 12

## Argomento: alberi binari + liste e code

Per questa esercitazione vengono forniti i file

- esercizio.h ed esercizio.c, contenenti la definizione e l'implementazione dei tipi di dato **TipoAlbero**, **TipoListaSCL** e **Coda**
- test\_esercizio.c, contiene un main di prova, che potete modificare a vostro piacimento per testare gli esercizi proposti.

```
#include "esercizio.h"

///////////////////////////////
////////// Alberi
///////////////////////////////

TipoAlbero albBinVuoto () {
    return 0;
}

TipoAlbero creaAlbBin( TipoInfoAlbero infoRadice, TipoAlbero sx, TipoAlbero dx ) {
    TipoAlbero a = ( TipoAlbero ) malloc ( sizeof (TipoNodoAlbero) ) ;
    a->info = infoRadice ;
    a->sinistro = sx ;
    a->destro = dx ;
    return a ;
}

bool estVuoto ( TipoAlbero a ) {
    return ( a == NULL );
}

TipoInfoAlbero radice ( TipoAlbero a ) {
    if ( a == NULL ) {
```

```

        printf ( " ERRORE accesso albero vuoto \n " );
        return ERRORE_InfoAlbero;
    }
    else
        return a->info;
}

TipoAlbero sinistro ( TipoAlbero a ) {
    if ( a == NULL ) {
        printf ( " ERRORE accesso albero vuoto \n " );
        return NULL;
    }
    else
        return a->sinistro ;
}

TipoAlbero destro ( TipoAlbero a ) {
    if ( a == NULL ) {
        printf ( " ERRORE accesso albero vuoto \n " );
        return NULL;
    }
    else
        return a->destro;
}

// Funzioni esterne

// Dealloca albero
void dealloca(TipoAlbero *a) {
    if (*a!=NULL) {
        dealloca(&((*a)->sinistro));
        dealloca(&((*a)->destro));
        free(*a);
        *a = NULL; // side-effect su a
    }
}

// Stampa
void stampaParentetica(TipoAlbero a) {
    if (estVuoto(a)) {

```

```

        printf("()");
    }
    else {
        printf("( %d ", radice(a));
        stampaParentetica(sinistro(a));
        printf(" ");
        stampaParentetica(destro(a));
        printf(" )");
    }
}

void stampaAlbero(const char* astr, TipoAlbero a) {
    printf("Albero %s: ",astr);
    stampaParentetica(a);
    printf("\n");
}

void disegnaAlbero_r(TipoAlbero a, char *s) {
    if (estVuoto(a))
        return;
    else {
        printf("    %s %d\n", s, radice(a));
        int l = strlen(s);
        s[l+1]='\0';
        s[l]='s';
        disegnaAlbero_r(sinistro(a), s);
        s[l]='d';
        disegnaAlbero_r(destro(a), s);
        s[l]='\0';
    }
}

void disegnaAlbero(const char* astr, TipoAlbero a) {
    char s[20]; // max profondità
    s[0]='\0';
    printf("Albero %s:\n",astr);
    disegnaAlbero_r(a,s);
    printf("\n");
}

```

```

// Lettura da file

void leggiParentesi(FILE *file_albero) {
    char c = ' ';
    while (c != '(' && c != ')')
        c = fgetc(file_albero);
}

TipoAlbero leggiSottoAlbero(FILE *file_albero) {
    char c;
    TipoInfoAlbero i;
    TipoAlbero r;

    leggiParentesi(file_albero); /* legge la parentesi aperta */
    c = fgetc(file_albero); /* carattere successivo */
    if (c == ')')
        return albBinVuoto(); /* se legge () allora l'albero e' vuoto */
    else {
        fscanf(file_albero, "%d", &i); /* altrimenti legge la radice */

        /* legge i sottoalberi */
        TipoAlbero s = leggiSottoAlbero(file_albero);
        TipoAlbero d = leggiSottoAlbero(file_albero);

        r = creaAlbBin(i,s,d);

        leggiParentesi(file_albero); /* legge la parentesi chiusa */

        return r;
    }
}

TipoAlbero leggiAlbero(char *nome_file) {
    TipoAlbero result;
    FILE *file_albero;
    file_albero = fopen(nome_file, "r");
    if (file_albero==NULL) {
        printf("File %s non trovato.\n", nome_file);
        return NULL;
    }
    result = leggiSottoAlbero(file_albero);
}

```

```

fclose(file_albero);
    return result;
}

TipoAlbero randomAlbero(int size) {
    if (size==0)
        return albBinVuoto();
    else if (size==1) {
        TipoInfoAlbero c = rand()%10;
        return creaAlbBin(c,albBinVuoto(),albBinVuoto());
    }
    else {
        TipoInfoAlbero c = rand()%10;
        int l = rand()%(size-1);
        int r = size - 1 - l;
        return creaAlbBin(c,randomAlbero(l),randomAlbero(r));
    }
}

///////////////////////////////
////////// Liste
/////////////////////////////
Lista listaVuota() {
    return NULL;
}

bool estVuota(Lista p) {
    return p == NULL;
}

Lista cons(InfoLista e, Lista p) {
    NodoListaSCL *n = (NodoListaSCL *)malloc(sizeof(NodoListaSCL));
    n->info = e;
    n->next = p;
    return n;
}

Lista cdr(Lista p) {
    return p->next;
}

```

```

InfoLista car(Lista p) {
    return p->info;
}

void stampaInfoLista(InfoLista i) {
    printf("%d ",i);
}

void stampaLista(Lista l) {
    NodoListaSCL *t = l;
    while (t!=NULL) {
        stampaInfoLista(t->info);
        t = t->next;
    }
    printf("\n");
}

Lista append(Lista l1, Lista l2) {
    if (estVuota(l1))
        return l2;
    else
        return cons(car(l1),append(cdr(l1),l2));
}

int lunghezzaLista(Lista l){
    int lun = 0;
    if(l == NULL)
        return 0;
    lun = 1;
    Lista aux = l;
    while(aux->next != NULL){
        aux = aux->next;
        lun++;
    }
    return lun;
}

///////////
////////// Code

```

```
//////////
```

```
Coda* codaVuota() {
    Coda* q = (Coda *)malloc(sizeof(Coda));
    NodoCodaSCL *n = (NodoCodaSCL *)malloc(sizeof(NodoCodaSCL)); // nodo generatore
    n->next = NULL;
    q->head = n;
    q->tail = n;
    return q;
}

bool estVuota(Coda *q) {
    return q->head->next == NULL;
}

void inCoda(Coda *q, InfoCoda e) { // inserimento in coda
    NodoCodaSCL *n = (NodoCodaSCL *)malloc(sizeof(NodoCodaSCL));
    n->info = e;
    n->next = NULL;
    q->tail->next = n;
    q->tail = n;
}

void outCoda(Coda *q) { // eliminazione in testa
    NodoCodaSCL *InfoCoda = q->head->next;
    if (InfoCoda!=NULL) {
        q->head->next = q->head->next->next;
        free(InfoCoda);
    }
    if (estVuota(q)) // coda vuota -> reset tail
        q->tail = q->head;
}

InfoCoda primo(Coda *q) {
    InfoCoda r = CODAERROR;
    if (q->head->next!=NULL) {
        r = q->head->next->info;
    }
    return r;
}

void stampaInfoCoda(InfoCoda i) {
```

```

    printf("%d ",i);
}

void stampaCoda(Coda *q) {
    NodoCodaSCL *t = q->head->next;
    while (t!=NULL) {
        stampaInfoCoda(t->info);
        t = t->next;
    }
    printf("\n");
}

```

## Esercizio 12.1

Implementare la funzione C

```
int ricercaLivello(TipoAlbero a, TipoInfoAlbero v, int livello);
```

che, dati un albero binario **a** e un valore **v**, restituisca il livello dell'albero dove si trova **v**.

NB: **livello** è il livello corrente dell'albero

```

int ricercaLivello(TipoAlbero a, TipoInfoAlbero v, int livello_corrente) {

    int livello = -1;
    if (a == NULL)
        return livello;
    if (a->info == v) {
        return livello_corrente;
    }
    livello = ricercaLivello(a->sinistro, v, livello_corrente+1);
    if (livello != -1)
        return livello;
    livello = ricercaLivello(a->destro, v, livello_corrente+1);
    if (livello != -1)
        return livello;

    return livello;
}

```

## Esercizio 12.2

Implementare la funzione C

```
int verificaNodi(TipoAlbero a, int livello);
```

Che, dato albero binario, restituisca la somma del numero di nodi di valore pari che si trovano nei livelli pari e di quello dei nodi di valore dispari che si trovano nei livelli dispari.

```
int verificaNodi(TipoAlbero a, int livello){

    int livello_corrente = livello;
    if( a==NULL )
        return 0;

    if( !(livello%2) ){
        if(!(a->info % 2))
            return 1 + verificaNodi(a->sinistro,livello+1) + verificaNodi(a->destro,livello+1);
        else
            return verificaNodi(a->sinistro,livello+1) + verificaNodi(a->destro,livello+1);
    } else if ( (livello%2) ){
        if( (a->info % 2) )
            return 1 + verificaNodi(a->sinistro,livello+1) + verificaNodi(a->destro,livello+1);
        else
            return verificaNodi(a->sinistro,livello+1) + verificaNodi(a->destro,livello+1);
    }
}
```

## Esercizio 12.3

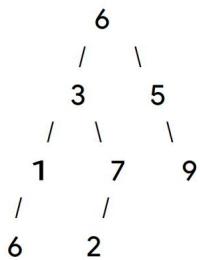
Implementare la seguente funzione C

```
int singleChildSum(TipoAlbero a);
```

che, dato in input un albero binario, restituisca la somma dei valori dei nodi che hanno un solo successore.

### Esempio

Dato il seguente albero binario **src** in ingresso:



`singleChildSum( src )` dovrà restituire:  $1 + 7 + 5 = 13$

```
int singleChildSum(TipoAlbero a){

    if( a==NULL )
        return 0;

    if( (a->sinistro && !a->destro) || (!a->sinistro && a->destro) )
        return 1 + singleChildSum(a->destro) + singleChildSum(a->sinistro);

    return singleChildSum(a->destro) + singleChildSum(a->sinistro);
}
```

## Esercizio 12.4

Implementare la seguente funzione C

```
Lista listaNodiFoglia(TipoAlbero a);
```

Che, dato un albero binario, restituisca una lista con i valori contenuti nelle foglie.

```
Lista listaNodiFoglia(TipoAlbero a){
```

```

Lista listaFoglie = listaVuota();
listaFoglie = listaNodiFoglia_aux(a, listaFoglie);
return listaFoglie;
}

Lista listaNodiFoglia_aux(TipoAlbero a, Lista l){

    if( a == NULL)
        return l;

    if( a->sinistro == NULL && a->destro == NULL ){
        l = cons(a->info, l);
        return l;
    }

    l = listaNodiFoglia_aux(a->sinistro, l);
    l = listaNodiFoglia_aux(a->destro, l);

    return l;
}

```

## Esercizio 12.5

Implementare la seguente funzione C

**Coda\* codaNodiDueFigli(TipoAlbero a);**

Che, dato un albero binario, restituisca una coda contenente i valori dei nodi dell'albero che hanno esattamente due figli.

```

Coda* codaNodiDueFigli(TipoAlbero a){
    Coda* coda = codaVuota();
    coda = codaNodiDueFigli_aux(a, coda);
    return coda;
}

Coda* codaNodiDueFigli_aux(TipoAlbero a, Coda* c){
    if( a == NULL)

```

```
    return c;

    if( a->sinistro != NULL && a->destro != NULL )
        inCoda(c, a->info);

    c = codaNodiDueFigli_aux(a->sinistro, c);
    c = codaNodiDueFigli_aux(a->destro, c);
    return c;
}
```

## Esercizio 12.6

Implementare la seguente funzione C

```
Lista listaPercorso(TipoAlbero a);
```

Che, dato un albero binario, restituisca la lista dei nodi contenuti nel percorso più lungo dalla radice a una delle foglie. Se esistono diversi percorsi di dimensione massima, si restituisca quello più a sinistra.

```
Lista listaPercorso(TipoAlbero a){

    if( a == NULL)
        return NULL;

    Lista l_left = listaPercorso(a->sinistro);
    Lista l_right = listaPercorso(a->destro);

    if (lunghezzaLista(l_left) >= lunghezzaLista(l_right) )
        return cons( a->info, l_left);
    else
        return cons(a->info, l_right);

}
```

## Esercizio 12.7

Implementare la seguente funzione C

```
void aggiungiFratello(TipoAlbero a);
```

Che, dato un albero binario, aggiunga una foglia ad ogni nodo avente solo un nodo figlio.  
Il valore del nuovo nodo deve essere 0.

```
void aggiungiFratello(TipoAlbero a){
    if( a==NULL )
        return;

    if(a->sinistro && a->destro)
    {
```

```
aggiungiFratello(a->sinistro);
aggiungiFratello(a->destro);
}
else if(a->sinistro && !a->destro)
{
    a->destro = (TipoAlbero) malloc(sizeof(TipoAlbero));
    a->destro->info = 0;
    aggiungiFratello(a->sinistro);
}
else if(!a->sinistro && a->destro)
{
    a->sinistro = (TipoAlbero) malloc(sizeof(TipoAlbero));
    a->sinistro->info = 0;
    aggiungiFratello(a->destro);
}
}
```