CATEGORIA 1: LISTE COLLEGATE

ESERCIZIO 1.1 - Clone Lista (ESAME 29/8/22)

Consegna originale: Scrivere la funzione ricorsiva `clone_list` che, ricevuta una lista collegata con puntatori, dovrà crearne una copia/clone.

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

struct nodo {
    float value;
    struct nodo *nextPtr;
};

typedef struct nodo Lista;

void clone_list(Lista *srcPtr, Lista **destPtr);
```

IMPLEMENTAZIONE:

```
void clone_list(Lista *srcPtr, Lista **destPtr) {
    // Caso base: lista vuota
    if (srcPtr == NULL) {
        *destPtr = NULL;
        return;
    }

    // Alloca nuovo nodo
    *destPtr = (Lista*)malloc(sizeof(Lista));
    if (*destPtr == NULL) {
        printf("Errore allocazione memoria\n");
        return;
    }

    // Copia il valore
    (*destPtr)->value = srcPtr->value;

    // Clone ricorsivo del resto della lista
    clone_list(srcPtr->nextPtr, &((*destPtr)->nextPtr));
}
```

ESERCIZIO 1.2 - Inserimento Ordinato Lista

Consegna: Implementa una funzione che inserisce un elemento in una lista ordinata mantenendo l'ordine.

IMPLEMENTAZIONE:

```
typedef struct nodo {
    int data;
    struct nodo *next;
} Nodo;

Nodo* inserisci_ordinato(Nodo *head, int valore) {
    // Crea nuovo nodo
    Nodo *nuovo = (Nodo*)malloc(sizeof(Nodo));
    if (nuovo == NULL) return head;

    nuovo->data = valore;
    nuovo->next = NULL;

    // Lista vuota o inserimento in testa
    if (head == NULL || head->data > valore) {
        nuovo->next = head;
        return nuovo;
    }

    // Trova posizione di inserimento
    Nodo *corrente = head;
    while (corrente->next != NULL && corrente->next->data < valore) {
        corrente = corrente->next;
    }

    // Inserisce il nodo
    nuovo->next = corrente->next;
    corrente->next = nuovo;

    return head;
}
```

ESERCIZIO 1.3 - Rimozione Duplicati

Consegna: Rimuovi tutti i duplicati da una lista ordinata.

IMPLEMENTAZIONE:

```
Nodo* rimuovi_duplicati(Nodo *head) {
    if (head == NULL) return NULL;

    Nodo *corrente = head;
```

```
    while (corrente->next != NULL) {
        if (corrente->data == corrente->next->data) {
            Nodo *duplicato = corrente->next;
            corrente->next = corrente->next->next;
            free(duplicato);
        } else {
            corrente = corrente->next;
        }
    }

    return head;
}
```

CATEGORIA 2: ALBERI BINARI DI RICERCA

ESERCIZIO 2.1 - Inserimento Ordinato BST (ESAME 29/8/22)

Consegna originale: Scrivere la funzione ricorsiva `ord_insert` che effettua l'inserimento di nuovi nodi in un albero binario di ricerca mantenendo l'ordine.

```
struct btree {
    int value;
    struct btree *leftPtr;
    struct btree *rightPtr;
};

typedef struct btree BTree;

void ord_insert(BTree **ptrPtr, int val);
```

IMPLEMENTAZIONE:

```
void ord_insert(BTree **ptrPtr, int val) {
    // Caso base: albero vuoto o posizione trovata
    if (*ptrPtr == NULL) {
        *ptrPtr = (BTree*)malloc(sizeof(BTree));
        if (*ptrPtr == NULL) {
            printf("Errore allocazione memoria\n");
            return;
        }
        (*ptrPtr)->value = val;
        (*ptrPtr)->leftPtr = NULL;
        (*ptrPtr)->rightPtr = NULL;
        return;
    }
```

```
    // Ricorsione: scegli sottoalbero appropriato
    if (val < (*ptrPtr)->value) {
        ord_insert(&((*ptrPtr)->leftPtr), val);
    } else if (val > (*ptrPtr)->value) {
        ord_insert(&((*ptrPtr)->rightPtr), val);
    }
    // Se val == (*ptrPtr)->value, non inserire (evita duplicati)
}
```

ESERCIZIO 2.2 - Visite dell'Albero

Consegna: Implementa le tre visite principali di un BST.

IMPLEMENTAZIONI:

```
// Visita simmetrica (in-order) - stampa valori ordinati
void visita_simmetrica(BTree *root) {
    if (root != NULL) {
        visita_simmetrica(root->leftPtr);
        printf("%d ", root->value);
        visita_simmetrica(root->rightPtr);
    }
}

// Visita anticipata (pre-order)
void visita_anticipata(BTree *root) {
    if (root != NULL) {
        printf("%d ", root->value);
        visita_anticipata(root->leftPtr);
        visita_anticipata(root->rightPtr);
    }
}

// Visita posticipata (post-order)
void visita_posticipata(BTree *root) {
    if (root != NULL) {
        visita_posticipata(root->leftPtr);
        visita_posticipata(root->rightPtr);
        printf("%d ", root->value);
    }
}
```

ESERCIZIO 2.3 - Ricerca in BST

Consegna: Implementa la ricerca di un elemento in un BST.

IMPLEMENTAZIONE:

```c
BTree* ricerca_bst(BTree *root, int valore) {
    // Caso base: albero vuoto o elemento trovato
    if (root == NULL || root->value == valore) {
        return root;
    }

    // Ricerca ricorsiva
    if (valore < root->value) {
        return ricerca_bst(root->leftPtr, valore);
    } else {
        return ricerca_bst(root->rightPtr, valore);
    }
}
```

ESERCIZIO 2.4 - Altezza Albero

Consegna: Calcola l'altezza di un albero binario.

IMPLEMENTAZIONE:

```c
int altezza_albero(BTree *root) {
    if (root == NULL) {
        return -1; // Convenzione: altezza albero vuoto = -1
    }

    int altezza_sx = altezza_albero(root->leftPtr);
    int altezza_dx = altezza_albero(root->rightPtr);

    return 1 + ((altezza_sx > altezza_dx) ? altezza_sx : altezza_dx);
}
```

CATEGORIA 3: MATRICI E ARRAY BIDIMENSIONALI

ESERCIZIO 3.1 - Mosse Alfiere (ESAME 29/8/22)

Consegna originale: Implementare una funzione che, a partire da una posizione nella scacchiera, segni tutte le mosse possibili per un alfiere. L'alfiere può spostarsi di un qualsiasi numero di caselle in diagonale.

IMPLEMENTAZIONE:

```c
#define SIZE 8

void mossa_alfiere(int scacchiera[SIZE][SIZE], int x, int y) {
```

```
    // Inizializza scacchiera a 0
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            scacchiera[i][j] = 0;
        }
    }

    // Controlla se posizione è valida
    if (x < 0 || x >= SIZE || y < 0 || y >= SIZE) {
        return; // Posizione non valida
    }

    // Direzioni diagonali: NE, NW, SE, SW
    int dir_x[] = {-1, -1, 1, 1};
    int dir_y[] = {1, -1, 1, -1};

    // Esplora tutte e 4 le diagonali
    for (int d = 0; d < 4; d++) {
        int nx = x + dir_x[d];
        int ny = y + dir_y[d];

        // Continua nella direzione finché rimani nella scacchiera
        while (nx >= 0 && nx < SIZE && ny >= 0 && ny < SIZE) {
            scacchiera[nx][ny] = 1;
            nx += dir_x[d];
            ny += dir_y[d];
        }
    }
}
```

ESERCIZIO 3.2 - Percorso nel Campo Fiorito

Consegna: Funzione ricorsiva che determina se esiste un percorso per attraversare un campo fiorito dal basso verso l'alto senza calpestare fiori (0=fiore, 1=libero). Mosse possibili: su o destra.

IMPLEMENTAZIONE:

```
#define RIGHE 5
#define COLONNE 5

int percorso_campo(int campo[RIGHE][COLONNE], int x, int y, int
visitato[RIGHE][COLONNE]) {
    // Caso base: raggiunta la riga superiore
    if (x == 0) {
        return 1;
    }
```

```
    // Fuori dai limiti o cella con fiore o già visitata
    if (x < 0 || x >= RIGHE || y < 0 || y >= COLONNE ||
        campo[x][y] == 0 || visitato[x][y] == 1) {
        return 0;
    }

    // Marca come visitato
    visitato[x][y] = 1;

    // Prova a muoverti verso l'alto o verso destra
    int risultato = percorso_campo(campo, x-1, y, visitato) ||
                    percorso_campo(campo, x, y+1, visitato);

    // Backtrack: rimuovi la marca (per altri percorsi)
    visitato[x][y] = 0;

    return risultato;
}

// Funzione wrapper
int esiste_percorso(int campo[RIGHE][COLONNE], int start_x, int start_y) {
    int visitato[RIGHE][COLONNE] = {0}; // Inizializza tutto a 0
    return percorso_campo(campo, start_x, start_y, visitato);
}
```

ESERCIZIO 3.3 - Percorsi su Griglia

Consegna: Calcola il numero di percorsi diversi dall'angolo in alto a sinistra a quello in basso a destra. Mosse: solo destra o giù.

IMPLEMENTAZIONE:

```
int conta_percorsi(int righe, int colonne, int x, int y) {
    // Caso base: raggiunta destinazione
    if (x == righe-1 && y == colonne-1) {
        return 1;
    }

    // Fuori dai limiti
    if (x >= righe || y >= colonne) {
        return 0;
    }

    // Somma percorsi andando giù e destra
    return conta_percorsi(righe, colonne, x+1, y) +
           conta_percorsi(righe, colonne, x, y+1);
}
```

CATEGORIA 4: FUNZIONI SU ARRAY

ESERCIZIO 4.1 - Verifica Array Tutti Pari (ESAME 29/8/22)

Consegna originale: Data la funzione, scrivere PRE e POST condizioni e dimostrarne la correttezza.

```
int f(int X[], int dim) {
    if (dim == 0)
        return 1;
    if (X[0] % 2 == 0)
        return f(X+1, dim-1);
    else
        return 0;
}
```

ANALISI:

- **PRE:** `dim >= 0` e `X` contiene almeno `dim` elementi validi
- **POST:** Restituisce 1 se tutti gli elementi di `X[0..dim-1]` sono pari, 0 altrimenti

DIMOSTRAZIONE CORRETTEZZA:

- **Caso base:** Se `dim == 0`, `f(X, dim) = 1` ed è vero che tutti gli elementi (nessuno) sono pari
- **Caso induttivo:** Se `X[0]` è dispari, ritorna 0 (corretto). Se `X[0]` è pari, il risultato dipende da `f(X+1, dim-1)` che per ipotesi induttiva è corretto per il resto dell'array

ESERCIZIO 4.2 - Rotazione Array

Consegna: Ruota un array di k posizioni verso sinistra.

IMPLEMENTAZIONE:

```
void ruota_sinistra(int arr[], int size, int k) {
    if (size <= 1) return;

    k = k % size; // Gestisce k > size
    if (k == 0) return;

    // Usa array temporaneo per semplicità
    int *temp = (int*)malloc(size * sizeof(int));
    if (temp == NULL) return;

    // Copia in posizione ruotata
```

```
    for (int i = 0; i < size; i++) {
        temp[i] = arr[(i + k) % size];
    }

    // Ricopia nell'array originale
    for (int i = 0; i < size; i++) {
        arr[i] = temp[i];
    }

    free(temp);
}
```

ESERCIZIO 4.3 - Merge Array Ordinati

Consegna: Fondi due array ordinati in un array risultante ordinato.

IMPLEMENTAZIONE:

```
void merge_arrays(int arr1[], int size1, int arr2[], int size2, int
result[]) {
    int i = 0, j = 0, k = 0;

    // Merge finché entrambi hanno elementi
    while (i < size1 && j < size2) {
        if (arr1[i] <= arr2[j]) {
            result[k++] = arr1[i++];
        } else {
            result[k++] = arr2[j++];
        }
    }

    // Copia elementi rimanenti di arr1
    while (i < size1) {
        result[k++] = arr1[i++];
    }

    // Copia elementi rimanenti di arr2
    while (j < size2) {
        result[k++] = arr2[j++];
    }
}
```

CATEGORIA 5: FUNZIONI GEOMETRICHE

ESERCIZIO 5.1 - Punto Interno/Esterno Rettangolo

Consegna: Determina se un punto (p_x, p_y) è interno o esterno a un rettangolo definito da due vertici opposti.

IMPLEMENTAZIONE:

```c
int interno_rettangolo(int s_x, int s_y, int d_x, int d_y, int p_x, int p_y)
{
    // Normalizza coordinate (assicura s < d)
    int min_x = (s_x < d_x) ? s_x : d_x;
    int max_x = (s_x > d_x) ? s_x : d_x;
    int min_y = (s_y < d_y) ? s_y : d_y;
    int max_y = (s_y > d_y) ? s_y : d_y;

    // Verifica se punto è interno (bordi esclusi)
    if (p_x > min_x && p_x < max_x && p_y > min_y && p_y < max_y) {
        return 1; // Interno
    } else {
        return 0; // Esterno o sul bordo
    }
}
```

CATEGORIA 6: RICORSIONE AVANZATA

ESERCIZIO 6.1 - Torre di Hanoi

Consegna: Risolvi il problema delle Torri di Hanoi.

IMPLEMENTAZIONE:

```c
void hanoi(int n, char sorgente, char destinazione, char ausiliario) {
    if (n == 1) {
        printf("Sposta disco da %c a %c\n", sorgente, destinazione);
        return;
    }

    // Sposta n-1 dischi da sorgente ad ausiliario
    hanoi(n-1, sorgente, ausiliario, destinazione);

    // Sposta il disco più grande
    printf("Sposta disco da %c a %c\n", sorgente, destinazione);

    // Sposta n-1 dischi da ausiliario a destinazione
    hanoi(n-1, ausiliario, destinazione, sorgente);
}
```

ESERCIZIO 6.2 - Potenza Efficiente

Consegna: Calcola a^n in modo efficiente.

IMPLEMENTAZIONE:

```
int potenza_veloce(int base, int esponente) {
    if (esponente == 0) return 1;
    if (esponente == 1) return base;

    if (esponente % 2 == 0) {
        int temp = potenza_veloce(base, esponente/2);
        return temp * temp;
    } else {
        return base * potenza_veloce(base, esponente-1);
    }
}
```
