CATEGORIA 1: LISTE COLLEGATE - VARIANTI

ESERCIZIO 1.1 - Lista Ordinata Decrescente

Consegna: Implementa una funzione che inserisce un elemento in una lista ordinata in modo **decrescente**.

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

Nodo* inserisci_decrescente(Nodo *head, int valore) {
    // TODO: Implementa qui
}
```

IMPLEMENTAZIONE:

```
Nodo* inserisci_decrescente(Nodo *head, int valore) {
    Nodo *nuovo = (Nodo*)malloc(sizeof(Nodo));
    if (nuovo == NULL) return head;

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

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

    // Trova posizione: cerca primo elemento minore di valore
    Nodo *corrente = head;
    while (corrente->next != NULL && corrente->next->data > valore) {
        corrente = corrente->next;
    }

    nuovo->next = corrente->next;
    corrente->next = nuovo;

    return head;
}
```

ESERCIZIO 1.2 - Fusione Liste con Duplicati

Consegna: Fondi due liste ordinate **eliminando i duplicati** durante la fusione.

IMPLEMENTAZIONE:

```
Nodo* fondi_senza_duplicati(Nodo *l1, Nodo *l2) {
    if (l1 == NULL) return l2;
    if (l2 == NULL) return l1;

    Nodo *risultato = NULL;
    Nodo **tail = &risultato;

    while (l1 != NULL && l2 != NULL) {
        int valore_da_inserire;

        if (l1->data < l2->data) {
            valore_da_inserire = l1->data;
            l1 = l1->next;
        } else if (l1->data > l2->data) {
            valore_da_inserire = l2->data;
            l2 = l2->next;
        } else {
            // Valori uguali: prendi uno solo
            valore_da_inserire = l1->data;
            l1 = l1->next;
            l2 = l2->next;
        }

        // Evita duplicati consecutivi
        if (risultato == NULL || (*tail)->data != valore_da_inserire) {
            *tail = (Nodo*)malloc(sizeof(Nodo));
            (*tail)->data = valore_da_inserire;
            (*tail)->next = NULL;
            tail = &((*tail)->next);
        }
    }

    // Aggiungi elementi rimanenti (evitando duplicati)
    Nodo *rimanente = (l1 != NULL) ? l1 : l2;
    while (rimanente != NULL) {
        if (risultato == NULL || (*tail)->data != rimanente->data) {
            *tail = (Nodo*)malloc(sizeof(Nodo));
            (*tail)->data = rimanente->data;
            (*tail)->next = NULL;
            tail = &((*tail)->next);
        }
        rimanente = rimanente->next;
    }
```

```
    return risultato;
}
```

ESERCIZIO 1.3 - Lista Circolare

Consegna: Implementa funzioni per una lista circolare: inserimento e verifica se è effettivamente circolare.

IMPLEMENTAZIONE:

```
typedef struct nodo_circ {
    int data;
    struct nodo_circ *next;
} NodoCirc;

// Inserisce in una lista circolare
NodoCirc* inserisci_circolare(NodoCirc *head, int valore) {
    NodoCirc *nuovo = (NodoCirc*)malloc(sizeof(NodoCirc));
    if (nuovo == NULL) return head;

    nuovo->data = valore;

    if (head == NULL) {
        // Lista vuota: nodo punta a se stesso
        nuovo->next = nuovo;
        return nuovo;
    }

    // Trova ultimo nodo (quello che punta a head)
    NodoCirc *ultimo = head;
    while (ultimo->next != head) {
        ultimo = ultimo->next;
    }

    // Inserisce il nuovo nodo
    nuovo->next = head;
    ultimo->next = nuovo;

    return head;
}

// Verifica se lista è circolare
int is_circolare(NodoCirc *head) {
    if (head == NULL) return 1; // Lista vuota è considerata circolare

    NodoCirc *lento = head;
    NodoCirc *veloce = head;
```

```
    do {
        lento = lento->next;
        veloce = veloce->next;
        if (veloce != NULL) veloce = veloce->next;

        if (veloce == NULL) return 0; // Non circolare

    } while (lento != veloce);

    return 1; // Circolare
}
```

CATEGORIA 2: ALBERI - VARIANTI AVANZATE

ESERCIZIO 2.1 - Conta Foglie in BST

Consegna: Conta il numero di foglie in un albero binario di ricerca.

IMPLEMENTAZIONE:

```
int conta_foglie(BTree *root) {
    if (root == NULL) {
        return 0;
    }

    // È una foglia se non ha figli
    if (root->leftPtr == NULL && root->rightPtr == NULL) {
        return 1;
    }

    return conta_foglie(root->leftPtr) + conta_foglie(root->rightPtr);
}
```

ESERCIZIO 2.2 - Trova Predecessore in BST

Consegna: Trova il predecessore di un valore in un BST (il valore immediatamente minore).

IMPLEMENTAZIONE:

```
BTree* trova_max(BTree *root) {
    if (root == NULL) return NULL;

    while (root->rightPtr != NULL) {
        root = root->rightPtr;
    }
```

```
    return root;
}

BTree* predecessore_bst(BTree *root, int valore) {
    if (root == NULL) return NULL;

    BTree *predecessore = NULL;
    BTree *corrente = root;

    while (corrente != NULL) {
        if (valore <= corrente->value) {
            corrente = corrente->leftPtr;
        } else {
            predecessore = corrente;
            corrente = corrente->rightPtr;
        }
    }

    return predecessore;
}
```

ESERCIZIO 2.3 - Albero Bilanciato

Consegna: Verifica se un albero binario è bilanciato (la differenza di altezza tra sottoalberi è al massimo 1).

IMPLEMENTAZIONE:

```
int altezza_bilanciato(BTree *root) {
    if (root == NULL) return 0;

    int altezza_sx = altezza_bilanciato(root->leftPtr);
    if (altezza_sx == -1) return -1; // Non bilanciato

    int altezza_dx = altezza_bilanciato(root->rightPtr);
    if (altezza_dx == -1) return -1; // Non bilanciato

    // Verifica bilanciamento
    if (abs(altezza_sx - altezza_dx) > 1) {
        return -1; // Non bilanciato
    }

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

int is_bilanciato(BTree *root) {
    return altezza_bilanciato(root) != -1;
}
```

CATEGORIA 3: MATRICI - PROBLEMI AVANZATI

ESERCIZIO 3.1 - Isola Massima

Consegna: Trova la dimensione dell'isola più grande in una matrice binaria (1=terra, 0=acqua). Isola = celle adiacenti con valore 1.

IMPLEMENTAZIONE:

```
#define RIGHE 5
#define COLONNE 5

int esplora_isola(int matrice[RIGHE][COLONNE], int x, int y, int
visitato[RIGHE][COLONNE]) {
    // Controlli di bounds e validità
    if (x < 0 || x >= RIGHE || y < 0 || y >= COLONNE ||
        matrice[x][y] == 0 || visitato[x][y] == 1) {
        return 0;
    }

    visitato[x][y] = 1;

    // Esplora le 4 direzioni adiacenti
    int dimensione = 1;
    dimensione += esplora_isola(matrice, x-1, y, visitato); // Su
    dimensione += esplora_isola(matrice, x+1, y, visitato); // Giù
    dimensione += esplora_isola(matrice, x, y-1, visitato); // Sinistra
    dimensione += esplora_isola(matrice, x, y+1, visitato); // Destra

    return dimensione;
}

int isola_massima(int matrice[RIGHE][COLONNE]) {
    int visitato[RIGHE][COLONNE] = {0};
    int max_dimensione = 0;

    for (int i = 0; i < RIGHE; i++) {
        for (int j = 0; j < COLONNE; j++) {
            if (matrice[i][j] == 1 && visitato[i][j] == 0) {
                int dimensione_corrente = esplora_isola(matrice, i, j,
visitato);
                if (dimensione_corrente > max_dimensione) {
                    max_dimensione = dimensione_corrente;
                }
            }
        }
    }
```

```
    return max_dimensione;
}
```

ESERCIZIO 3.2 - Spirale Matrice

IMPLEMENTAZIONE:

```
void stampa_spirale(int matrice[][4], int righe, int colonne) {
    int top = 0, bottom = righe - 1;
    int left = 0, right = colonne - 1;

    while (top <= bottom && left <= right) {
        // Stampa riga superiore (sinistra → destra)
        for (int j = left; j <= right; j++) {
            printf("%d ", matrice[top][j]);
        }
        top++;

        // Stampa colonna destra (alto → basso)
        for (int i = top; i <= bottom; i++) {
            printf("%d ", matrice[i][right]);
        }
        right--;

        // Stampa riga inferiore (destra → sinistra)
        if (top <= bottom) {
            for (int j = right; j >= left; j--) {
                printf("%d ", matrice[bottom][j]);
            }
            bottom--;
        }

        // Stampa colonna sinistra (basso → alto)
        if (left <= right) {
            for (int i = bottom; i >= top; i--) {
                printf("%d ", matrice[i][left]);
            }
            left++;
        }
    }
}
```

ESERCIZIO 3.3 - Rotazione Matrice 90°

Consegna: Ruota una matrice quadrata di 90 gradi in senso orario **in place**.

IMPLEMENTAZIONE:

```c
void ruota_90_gradi(int matrice[][4], int n) {
    // Prima: trasponi la matrice
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            int temp = matrice[i][j];
            matrice[i][j] = matrice[j][i];
            matrice[j][i] = temp;
        }
    }

    // Seconda: inverti ogni riga
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n/2; j++) {
            int temp = matrice[i][j];
            matrice[i][j] = matrice[i][n-1-j];
            matrice[i][n-1-j] = temp;
        }
    }
}
```

ESERCIZIO 3.4 - Cammino Minimo con Ostacoli

Consegna: Trova il cammino minimo in una griglia con ostacoli (0=libero, 1=ostacolo).

IMPLEMENTAZIONE:

```c
#include <limits.h>

int cammino_minimo_rec(int griglia[][4], int righe, int colonne, int x, int
y, int memo[][4]) {
    // Fuori dai limiti o ostacolo
    if (x >= righe || y >= colonne || griglia[x][y] == 1) {
        return INT_MAX;
    }

    // Destinazione raggiunta
    if (x == righe-1 && y == colonne-1) {
        return 1;
    }

    // Se già calcolato, ritorna valore memorizzato
    if (memo[x][y] != -1) {
        return memo[x][y];
    }

    // Calcola cammino minimo andando giù o destra
    int giu = cammino_minimo_rec(griglia, righe, colonne, x+1, y, memo);
    int destra = cammino_minimo_rec(griglia, righe, colonne, x, y+1, memo);
```

```
    int minimo = (giu < destra) ? giu : destra;
    memo[x][y] = (minimo == INT_MAX) ? INT_MAX : minimo + 1;

    return memo[x][y];
}

int cammino_minimo(int griglia[][4], int righe, int colonne) {
    int memo[4][4];
    for (int i = 0; i < righe; i++) {
        for (int j = 0; j < colonne; j++) {
            memo[i][j] = -1;
        }
    }

    int risultato = cammino_minimo_rec(griglia, righe, colonne, 0, 0, memo);
    return (risultato == INT_MAX) ? -1 : risultato;
}
```

CATEGORIA 4: STRINGHE E CARATTERI

ESERCIZIO 4.1 - Palindromo con Caratteri Speciali

Consegna: Verifica se una stringa è palindroma ignorando spazi e punteggiatura, considerando solo lettere.

IMPLEMENTAZIONE:

```
#include <ctype.h>

int is_palindromo_avanzato(char str[]) {
    int lunghezza = strlen(str);
    int inizio = 0, fine = lunghezza - 1;

    while (inizio < fine) {
        // Salta caratteri non alfabetici da sinistra
        while (inizio < fine && !isalpha(str[inizio])) {
            inizio++;
        }

        // Salta caratteri non alfabetici da destra
        while (inizio < fine && !isalpha(str[fine])) {
            fine--;
        }

        // Confronta caratteri (case-insensitive)
```

```
        if (tolower(str[inizio]) != tolower(str[fine])) {
            return 0;
        }

        inizio++;
        fine--;
    }

    return 1;
}
```

ESERCIZIO 4.2 - Anagrammi

Consegna: Verifica se due stringhe sono anagrammi (stesse lettere, ordine diverso).

IMPLEMENTAZIONE:

```
int sono_anagrammi(char str1[], char str2[]) {
    int freq1[26] = {0}; // Frequenza caratteri str1
    int freq2[26] = {0}; // Frequenza caratteri str2

    int len1 = strlen(str1);
    int len2 = strlen(str2);

    // Lunghezze diverse = non anagrammi
    if (len1 != len2) return 0;

    // Conta frequenze (considera solo lettere minuscole)
    for (int i = 0; i < len1; i++) {
        if (isalpha(str1[i])) {
            freq1[tolower(str1[i]) - 'a']++;
        }
        if (isalpha(str2[i])) {
            freq2[tolower(str2[i]) - 'a']++;
        }
    }

    // Confronta frequenze
    for (int i = 0; i < 26; i++) {
        if (freq1[i] != freq2[i]) {
            return 0;
        }
    }

    return 1;
}
```

ESERCIZIO 4.3 - Compressione Stringa

Consegna: Comprimi una stringa sostituendo caratteri ripetuti con carattere+conteggio (es: "aaabbc" → "a3b2c1").

IMPLEMENTAZIONE:

```
void comprimi_stringa(char input[], char output[]) {
    int len = strlen(input);
    if (len == 0) {
        output[0] = '\0';
        return;
    }

    int output_index = 0;
    int count = 1;

    for (int i = 1; i <= len; i++) { // <= per gestire ultimo gruppo
        if (i < len && input[i] == input[i-1]) {
            count++;
        } else {
            // Fine gruppo: scrivi carattere e conteggio
            output[output_index++] = input[i-1];

            // Converte conteggio in stringa
            if (count < 10) {
                output[output_index++] = '0' + count;
            } else {
                // Gestisce conteggi > 9 (opzionale)
                sprintf(&output[output_index], "%d", count);
                output_index += strlen(&output[output_index]);
            }

            count = 1;
        }
    }

    output[output_index] = '\0';
}
```

CATEGORIA 5: RICORSIONE CON BACKTRACKING

ESERCIZIO 5.1 - N-Regine

Consegna: Risolvi il problema delle N-Regine: posiziona N regine su una scacchiera N×N senza che si attacchino.

IMPLEMENTAZIONE:

```c
int is_sicuro(int scacchiera[][8], int riga, int col, int n) {
    // Controlla colonna
    for (int i = 0; i < riga; i++) {
        if (scacchiera[i][col] == 1) {
            return 0;
        }
    }

    // Controlla diagonale superiore sinistra
    for (int i = riga-1, j = col-1; i >= 0 && j >= 0; i--, j--) {
        if (scacchiera[i][j] == 1) {
            return 0;
        }
    }

    // Controlla diagonale superiore destra
    for (int i = riga-1, j = col+1; i >= 0 && j < n; i--, j++) {
        if (scacchiera[i][j] == 1) {
            return 0;
        }
    }

    return 1;
}

int risolvi_n_regine(int scacchiera[][8], int riga, int n) {
    // Caso base: tutte le regine posizionate
    if (riga >= n) {
        return 1;
    }

    // Prova ogni colonna nella riga corrente
    for (int col = 0; col < n; col++) {
        if (is_sicuro(scacchiera, riga, col, n)) {
            scacchiera[riga][col] = 1;

            // Ricorsione per riga successiva
            if (risolvi_n_regine(scacchiera, riga + 1, n)) {
                return 1;
            }

            // Backtrack
            scacchiera[riga][col] = 0;
        }
    }

    return 0; // Nessuna soluzione
}
```

```c
void n_regine(int n) {
    int scacchiera[8][8] = {0};

    if (risolvi_n_regine(scacchiera, 0, n)) {
        printf("Soluzione trovata:\n");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                printf("%d ", scacchiera[i][j]);
            }
            printf("\n");
        }
    } else {
        printf("Nessuna soluzione esistente.\n");
    }
}
```

ESERCIZIO 5.2 - Sudoku Solver

Consegna: Risolvi un puzzle Sudoku 9×9 usando backtracking.

IMPLEMENTAZIONE:

```c
int is_valido_sudoku(int griglia[9][9], int riga, int col, int num) {
    // Verifica riga
    for (int j = 0; j < 9; j++) {
        if (griglia[riga][j] == num) {
            return 0;
        }
    }

    // Verifica colonna
    for (int i = 0; i < 9; i++) {
        if (griglia[i][col] == num) {
            return 0;
        }
    }

    // Verifica box 3×3
    int start_riga = riga - riga % 3;
    int start_col = col - col % 3;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (griglia[i + start_riga][j + start_col] == num) {
                return 0;
            }
        }
    }
```

```
    return 1;
}

int risolvi_sudoku(int griglia[9][9]) {
    for (int riga = 0; riga < 9; riga++) {
        for (int col = 0; col < 9; col++) {
            if (griglia[riga][col] == 0) {
                // Prova numeri da 1 a 9
                for (int num = 1; num <= 9; num++) {
                    if (is_valido_sudoku(griglia, riga, col, num)) {
                        griglia[riga][col] = num;

                        if (risolvi_sudoku(griglia)) {
                            return 1;
                        }

                        // Backtrack
                        griglia[riga][col] = 0;
                    }
                }
                return 0; // Nessun numero valido
            }
        }
    }
    return 1; // Sudoku risolto
}
```

CATEGORIA 6: ORDINAMENTO AVANZATO

ESERCIZIO 6.1 - Quick Sort con Pivot Mediano

Consegna: Implementa Quick Sort scegliendo come pivot il mediano tra primo, ultimo e centrale.

IMPLEMENTAZIONE:

```
int mediano_di_tre(int arr[], int basso, int alto) {
    int centro = basso + (alto - basso) / 2;

    if (arr[centro] < arr[basso]) {
        scambia(&arr[centro], &arr[basso]);
    }
    if (arr[alto] < arr[basso]) {
        scambia(&arr[alto], &arr[basso]);
    }
    if (arr[alto] < arr[centro]) {
```

```
        scambia(&arr[alto], &arr[centro]);
    }

    // Metti il mediano alla fine
    scambia(&arr[centro], &arr[alto]);
    return alto;
}

void scambia(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partiziona_mediano(int arr[], int basso, int alto) {
    int pivot_index = mediano_di_tre(arr, basso, alto);
    int pivot = arr[pivot_index];

    int i = basso - 1;

    for (int j = basso; j < alto; j++) {
        if (arr[j] < pivot) {
            i++;
            scambia(&arr[i], &arr[j]);
        }
    }

    scambia(&arr[i + 1], &arr[alto]);
    return i + 1;
}

void quick_sort_mediano(int arr[], int basso, int alto) {
    if (basso < alto) {
        int pi = partiziona_mediano(arr, basso, alto);

        quick_sort_mediano(arr, basso, pi - 1);
        quick_sort_mediano(arr, pi + 1, alto);
    }
}
```

ESERCIZIO 6.2 - Merge Sort Bottom-Up

Consegna: Implementa Merge Sort iterativo (bottom-up) invece che ricorsivo.

IMPLEMENTAZIONE:

```
void merge(int arr[], int sinistra, int centro, int destra) {
    int n1 = centro - sinistra + 1;
    int n2 = destra - centro;
```

```c
    int *L = (int*)malloc(n1 * sizeof(int));
    int *R = (int*)malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++) {
        L[i] = arr[sinistra + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[centro + 1 + j];
    }

    int i = 0, j = 0, k = sinistra;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);
}

void merge_sort_iterativo(int arr[], int n) {
    // Inizia con sottoaree di dimensione 1, poi 2, 4, 8...
    for (int curr_size = 1; curr_size < n; curr_size = 2 * curr_size) {
        // Prendi punto di partenza per sottoarea sinistra
        for (int left_start = 0; left_start < n - 1; left_start += 2 *
curr_size) {
            int centro = left_start + curr_size - 1;
            int right_end = (left_start + 2 * curr_size - 1 < n - 1) ?
                            left_start + 2 * curr_size - 1 : n - 1;
```

```
            if (centro < right_end) {
                merge(arr, left_start, centro, right_end);
            }
        }
    }
}
```

CATEGORIA 7: PROBLEMI MISTI AVANZATI

ESERCIZIO 7.1 - LCS (Longest Common Subsequence)

Consegna: Trova la lunghezza della più lunga sottosequenza comune tra due stringhe.

IMPLEMENTAZIONE:

```
int lcs_ricorsivo(char str1[], char str2[], int m, int n, int memo[][100]) {
    if (m == 0 || n == 0) {
        return 0;
    }

    if (memo[m][n] != -1) {
        return memo[m][n];
    }

    if (str1[m-1] == str2[n-1]) {
        memo[m][n] = 1 + lcs_ricorsivo(str1, str2, m-1, n-1, memo);
    } else {
        int lcs1 = lcs_ricorsivo(str1, str2, m, n-1, memo);
        int lcs2 = lcs_ricorsivo(str1, str2, m-1, n, memo);
        memo[m][n] = (lcs1 > lcs2) ? lcs1 : lcs2;
    }

    return memo[m][n];
}

int lcs(char str1[], char str2[]) {
    int m = strlen(str1);
    int n = strlen(str2);
    int memo[100][100];

    // Inizializza memo
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            memo[i][j] = -1;
        }
    }
```

```
    return lcs_ricorsivo(str1, str2, m, n, memo);
}
```

ESERCIZIO 7.2 - Knapsack 0/1

Consegna: Risolvi il problema dello zaino: massimizza il valore con peso limitato.

IMPLEMENTAZIONE:

```
int knapsack_rec(int pesi[], int valori[], int n, int capacita, int memo[]
[1001]) {
    if (n == 0 || capacita == 0) {
        return 0;
    }

    if (memo[n][capacita] != -1) {
        return memo[n][capacita];
    }

    // Se peso dell'oggetto > capacità, non può essere incluso
    if (pesi[n-1] > capacita) {
        memo[n][capacita] = knapsack_rec(pesi, valori, n-1, capacita, memo);
    } else {
        // Massimo tra includere e non includere l'oggetto
        int includi = valori[n-1] + knapsack_rec(pesi, valori, n-1, capacita
- pesi[n-1], memo);
        int escludi = knapsack_rec(pesi, valori, n-1, capacita, memo);
        memo[n][capacita] = (includi > escludi) ? includi : escludi;
    }

    return memo[n][capacita];
}

int knapsack(int pesi[], int valori[], int n, int capacita) {
    int memo[101][1001];

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= capacita; j++) {
            memo[i][j] = -1;
        }
    }

    return knapsack_rec(pesi, valori, n, capacita, memo);
}
```

SUGGERIMENTI PER L'ESAME

1. **Pattern Recognition:** Questi esercizi seguono pattern riconoscibili. Studia la struttura generale.
2. **Gestione Memoria:** Sempre controllare malloc/free e puntatori NULL.
3. **Casi Base:** In ricorsione, definire sempre chiaramente i casi base.
4. **Testing:** Testa sempre con casi edge (array vuoti, input NULL, ecc.).
5. **Complessità:** Conosci la complessità temporale e spaziale delle tue soluzioni.

Buono studio e buona fortuna all'esame!