CATEGORIA 1: LISTE COLLEGATE

1.1 CLONE_LIST - Clonazione di 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);
```

ALGORITMO RICORSIVO

```
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));
}
```

ALGORITMO ITERATIVO

```
void clone_list_iterativo(Lista *srcPtr, Lista **destPtr) {
    *destPtr = NULL;
    Lista **current = destPtr;

    while (srcPtr != NULL) {
        *current = malloc(sizeof(Lista));
        if (*current == NULL) {
            printf("Errore allocazione memoria\n");
            return;
        }

        (*current)->value = srcPtr->value;
        (*current)->nextPtr = NULL;

        current = &((*current)->nextPtr);
        srcPtr = srcPtr->nextPtr;
    }
}
```

1.2 INSERIMENTO ORDINATO LISTA

CONSEGNA

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

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

ALGORITMO ITERATIVO

```
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;
}
```

ALGORITMO RICORSIVO

```
Nodo* inserisci_ordinato_ricorsivo(Nodo *head, int valore) {
    // Caso base: lista vuota o inserimento in testa
    if (head == NULL || head->data > valore) {
        Nodo *nuovo = (Nodo*)malloc(sizeof(Nodo));
        nuovo->data = valore;
        nuovo->next = head;
        return nuovo;
    }

    // Ricorsione sul resto della lista
    head->next = inserisci_ordinato_ricorsivo(head->next, valore);
    return head;
}
```

1.3 RIMOZIONE DUPLICATI

CONSEGNA

Rimuovi tutti i duplicati da una lista ordinata.

ALGORITMO ITERATIVO

```
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;
}
```

ALGORITMO RICORSIVO

```
Nodo* rimuovi_duplicati_ricorsivo(Nodo *head) {
    if (head == NULL || head->next == NULL) return head;

    // Ricorsione sul resto
    head->next = rimuovi_duplicati_ricorsivo(head->next);

    // Se duplicato, elimina il nodo corrente
    if (head->data == head->next->data) {
        Nodo *temp = head;
        head = head->next;
        free(temp);
    }

    return head;
}
```

1.4 CLONE_INVLIST - Clonazione Lista Invertita

CONSEGNA

Scrivere la funzione `clone_invlist` che crea una copia della lista sorgente ma con ordine invertito.

ALGORITMO RICORSIVO

```
void clone_invlist_ricorsivo(Lista *srcPtr, Lista **destPtr) {
    if (srcPtr == NULL) {
        *destPtr = NULL;
        return;
    }

    // Ricorsione prima
    clone_invlist_ricorsivo(srcPtr->nextPtr, destPtr);

    // Inserimento in testa nella lista destinazione
    Lista *newNode = malloc(sizeof(Lista));
    newNode->value = srcPtr->value;
    newNode->nextPtr = *destPtr;
```

```
    *destPtr = newNode;
}
```

ALGORITMO ITERATIVO

```
void clone_invlist_iterativo(Lista *srcPtr, Lista **destPtr) {
    *destPtr = NULL;

    while (srcPtr != NULL) {
        Lista *newNode = malloc(sizeof(Lista));
        newNode->value = srcPtr->value;
        newNode->nextPtr = *destPtr;
        *destPtr = newNode;

        srcPtr = srcPtr->nextPtr;
    }
}
```

1.5 INTERSECTION - Intersezione tra Liste

CONSEGNA

Scrivere la funzione `intersection` che, date due liste, crea una terza lista contenente solo gli elementi presenti in entrambe.

ALGORITMO

```
void intersection(Lista *list1, Lista *list2, Lista **destPtr) {
    *destPtr = NULL;
    Lista **current = destPtr;

    while (list1 != NULL) {
        // Cerco se l'elemento di list1 è presente in list2
        Lista *temp = list2;
        int found = 0;

        while (temp != NULL && !found) {
            if (temp->valore == list1->valore) {
                found = 1;
            }
            temp = temp->nextPtr;
        }

        if (found) {
            *current = malloc(sizeof(Lista));
            (*current)->valore = list1->valore;
            (*current)->nextPtr = NULL;
```

```
            current = &((*current)->nextPtr);
        }

        list1 = list1->nextPtr;
    }
}
```

CATEGORIA 2: ALBERI BINARI DI RICERCA

2.1 ORD_INSERT - 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);
```

ALGORITMO RICORSIVO

```
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)
}
```

ALGORITMO ITERATIVO

```
void ord_insert_iterativo(BTree **root, int val) {
    BTree *nuovo = (BTree*)malloc(sizeof(BTree));
    nuovo->value = val;
    nuovo->leftPtr = nuovo->rightPtr = NULL;

    if (*root == NULL) {
        *root = nuovo;
        return;
    }

    BTree *current = *root;
    BTree *parent = NULL;

    while (current != NULL) {
        parent = current;
        if (val < current->value) {
            current = current->leftPtr;
        } else if (val > current->value) {
            current = current->rightPtr;
        } else {
            free(nuovo); // Duplicato
            return;
        }
    }

    if (val < parent->value) {
        parent->leftPtr = nuovo;
    } else {
        parent->rightPtr = nuovo;
    }
}
```

2.2 VISITE DELL'ALBERO

CONSEGNA

Implementa le tre visite principali di un BST.

VISITA SIMMETRICA (IN-ORDER) - RICORSIVA

```c
void visita_simmetrica(BTree *root) {
    if (root != NULL) {
        visita_simmetrica(root->leftPtr);
        printf("%d ", root->value);
        visita_simmetrica(root->rightPtr);
    }
}
```

VISITA ANTICIPATA (PRE-ORDER) - RICORSIVA

```c
void visita_anticipata(BTree *root) {
    if (root != NULL) {
        printf("%d ", root->value);
        visita_anticipata(root->leftPtr);
        visita_anticipata(root->rightPtr);
    }
}
```

VISITA POSTICIPATA (POST-ORDER) - RICORSIVA

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

VISITE ITERATIVE (CON STACK)

```c
#include <stdlib.h>

typedef struct {
    BTree **items;
    int top;
    int capacity;
} Stack;

void visita_simmetrica_iterativa(BTree *root) {
    if (root == NULL) return;

    Stack stack = {0};
    stack.capacity = 100;
    stack.items = malloc(stack.capacity * sizeof(BTree*));
    stack.top = -1;
```

```
    BTree *current = root;

    while (current != NULL || stack.top >= 0) {
        while (current != NULL) {
            stack.items[++stack.top] = current;
            current = current->leftPtr;
        }

        current = stack.items[stack.top--];
        printf("%d ", current->value);
        current = current->rightPtr;
    }

    free(stack.items);
}
```

2.3 RICERCA IN BST

CONSEGNA

Implementa la ricerca di un elemento in un BST.

ALGORITMO RICORSIVO

```
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);
    }
}
```

ALGORITMO ITERATIVO

```
BTree* ricerca_bst_iterativo(BTree *root, int valore) {
    while (root != NULL && root->value != valore) {
        if (valore < root->value) {
            root = root->leftPtr;
        } else {
            root = root->rightPtr;
        }
```

```
    }
    return root;
}
```

2.4 ALTEZZA ALBERO

CONSEGNA

Calcola l'altezza di un albero binario.

ALGORITMO RICORSIVO

```
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);
}
```

ALGORITMO ITERATIVO (BFS)

```
int altezza_albero_iterativo(BTree *root) {
    if (root == NULL) return -1;

    BTree *queue[1000];
    int front = 0, rear = 0;
    queue[rear++] = root;
    queue[rear++] = NULL; // Separatore livelli

    int altezza = 0;

    while (front < rear) {
        BTree *current = queue[front++];

        if (current == NULL) {
            if (front < rear) {
                queue[rear++] = NULL;
                altezza++;
            }
        } else {
            if (current->leftPtr) queue[rear++] = current->leftPtr;
            if (current->rightPtr) queue[rear++] = current->rightPtr;
        }
```

```
    }

    return altezza;
}
```

CATEGORIA 3: MATRICI E ARRAY BIDIMENSIONALI

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.

```
#define SIZE 8

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

ALGORITMO

```
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];
        }
    }
}
```

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.

```
#define RIGHE 5
#define COLONNE 5

int percorso_campo(int campo[RIGHE][COLONNE], int x, int y, int
visitato[RIGHE][COLONNE]);
```

ALGORITMO RICORSIVO

```
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);
}
```

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.

ALGORITMO RICORSIVO

```
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);
}
```

ALGORITMO DINAMICO (più efficiente)

```
int conta_percorsi_dp(int righe, int colonne) {
    int dp[righe][colonne];

    // Inizializza prima riga e prima colonna
    for (int i = 0; i < righe; i++) dp[i][0] = 1;
    for (int j = 0; j < colonne; j++) dp[0][j] = 1;

    // Riempi la tabella
    for (int i = 1; i < righe; i++) {
        for (int j = 1; j < colonne; j++) {
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
```

```
    }

    return dp[righe-1][colonne-1];
}
```

CATEGORIA 4: FUNZIONI SU ARRAY

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

VERSIONE ITERATIVA

```
int tutti_pari_iterativo(int X[], int dim) {
    for (int i = 0; i < dim; i++) {
        if (X[i] % 2 != 0) {
            return 0;
        }
    }
```

```
    return 1;
}
```

4.2 ROTAZIONE ARRAY

CONSEGNA

Ruota un array di k posizioni verso sinistra.

ALGORITMO CON ARRAY TEMPORANEO

```
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);
}
```

ALGORITMO IN-PLACE (Reverse)

```
void reverse(int arr[], int start, int end) {
    while (start < end) {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

void ruota_sinistra_inplace(int arr[], int size, int k) {
    k = k % size;
```

```
    if (k == 0) return;

    // Inversione di tutto l'array
    reverse(arr, 0, size-1);

    // Inversione delle prime (size-k) posizioni
    reverse(arr, 0, size-k-1);

    // Inversione delle ultime k posizioni
    reverse(arr, size-k, size-1);
}
```

4.3 MERGE ARRAY ORDINATI

CONSEGNA

Fondi due array ordinati in un array risultante ordinato.

ALGORITMO

```
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++];
    }
}
```

4.4 MASSIMO ARRAY (Dai compitini)

CONSEGNA

Trovare il massimo elemento di un array.

ALGORITMO RICORSIVO

```
int massimo_ricorsivo(int arr[], int dim) {
    if (dim == 1) {
        return arr[0];
    }

    int max_resto = massimo_ricorsivo(arr + 1, dim - 1);
    return (arr[0] > max_resto) ? arr[0] : max_resto;
}
```

ALGORITMO ITERATIVO

```
int massimo_iterativo(int arr[], int dim) {
    if (dim <= 0) return INT_MIN; // Valore sentinella

    int max = arr[0];
    for (int i = 1; i < dim; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

CATEGORIA 5: FUNZIONI GEOMETRICHE

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.

ALGORITMO

```
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
    }
}
```

VERSIONE CON BORDI INCLUSI

```
int interno_rettangolo_bordi(int s_x, int s_y, int d_x, int d_y, int p_x,
int p_y) {
    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;

    return (p_x >= min_x && p_x <= max_x && p_y >= min_y && p_y <= max_y);
}
```

CATEGORIA 6: RICORSIONE AVANZATA

6.1 TORRE DI HANOI

CONSEGNA

Risolvi il problema delle Torri di Hanoi.

ALGORITMO RICORSIVO

```
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);
}
```

VERSIONE CON CONTEGGIO MOSSE

```
int hanoi_conta_mosse(int n) {
    if (n == 1) return 1;
    return 2 * hanoi_conta_mosse(n-1) + 1;
}

// Formula diretta: 2^n - 1
int mosse_hanoi(int n) {
    return (1 << n) - 1; // 2^n - 1
}
```

6.2 POTENZA EFFICIENTE

CONSEGNA

Calcola a^n in modo efficiente.

ALGORITMO RICORSIVO (O(log n))

```
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);
    }
}
```

ALGORITMO ITERATIVO

```
int potenza_iterativo(int base, int esponente) {
    int risultato = 1;
    while (esponente > 0) {
        if (esponente % 2 == 1) {
            risultato *= base;
        }
        base *= base;
        esponente /= 2;
    }
```

```
    return risultato;
}
```

6.3 FIBONACCI

CONSEGNA

Calcola l'n-esimo numero di Fibonacci.

ALGORITMO RICORSIVO NAIVE (O(2^n))

```
int fibonacci_ricorsivo(int n) {
    if (n <= 1) return n;
    return fibonacci_ricorsivo(n-1) + fibonacci_ricorsivo(n-2);
}
```

ALGORITMO ITERATIVO (O(n))

```
int fibonacci_iterativo(int n) {
    if (n <= 1) return n;

    int a = 0, b = 1, temp;
    for (int i = 2; i <= n; i++) {
        temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}
```

6.4 FATTORIALE PRODOTTO (Dai compitini)

CONSEGNA

Calcola n *(n-1)* (n-2) ... m

ALGORITMO RICORSIVO

```
int fattoriale_prodotto(int n, int m) {
    // PRE: n >= m >= 1
    // POST: Calcola n * (n-1) * (n-2) * ... * m

    if (n == m) {
        return n;
    } else {
        return n * fattoriale_prodotto(n-1, m);
```

```
    }
}
```

ALGORITMO ITERATIVO

```
int fattoriale_prodotto_iterativo(int n, int m) {
    int result = 1;
    for (int i = n; i >= m; i--) {
        result *= i;
    }
    return result;
}
```

CATEGORIA 7: FUNZIONI SU STRINGHE

7.1 COPIA_STRINGA RICORSIVA (Dai compitini)

CONSEGNA

Scrivere una funzione ricorsiva che copia il contenuto della seconda stringa nella prima.

ALGORITMO RICORSIVO

```
void copia_stringa(char *s1, char *s2) {
    /*
    PRE: s1, s2 sono stringhe. L'array di caratteri contenente
         s1 è più lungo di s2.
    POST: copia il contenuto di s2 in s1
    */
    *s1 = *s2;
    if (*s2 != '\0')
        copia_stringa(s1+1, s2+1);
}
```

ALGORITMO ITERATIVO

```
void copia_stringa_iterativo(char *dest, char *src) {
    while (*src != '\0') {
        *dest = *src;
        dest++;
        src++;
    }
```

```
    *dest = '\0'; // Termina la stringa
}
```

7.2 LUNGHEZZA STRINGA

ALGORITMO RICORSIVO

```
int lunghezza_stringa(char *str) {
    if (*str == '\0') {
        return 0;
    } else {
        return 1 + lunghezza_stringa(str + 1);
    }
}
```

ALGORITMO ITERATIVO

```
int lunghezza_stringa_iterativo(char *str) {
    int count = 0;
    while (*str != '\0') {
        count++;
        str++;
    }
    return count;
}
```

CATEGORIA 8: FUNZIONI MATEMATICHE COMPLESSE

8.1 MINIMI_TERMINI - Riduzione Frazioni

CONSEGNA

Scrivere una funzione `minimi_termini()` che, dati due interi positivi `num` e `den`, riduce ai minimi termini la frazione `num/den`.

ALGORITMO

```
int gcd(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```

```c
void minimi_termini(int *num, int *den) {
    /*
    PRE: num > 0, den > 0
    POST: *num e *den rappresentano la frazione ridotta ai minimi termini
    */
    int divisore = gcd(*num, *den);
    *num = *num / divisore;
    *den = *den / divisore;
}

int main() {
    int numeratore = 6, denominatore = 12;
    printf("%d/%d=", numeratore, denominatore);
    minimi_termini(&numeratore, &denominatore);
    printf("%d/%d\n", numeratore, denominatore);
    return 0;
}
```

GCD ITERATIVO

```c
int gcd_iterativo(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

8.2 COMBINAZIONI

CONSEGNA

Calcola C(n,k) = n! / (k! * (n-k)!)

ALGORITMO RICORSIVO

```c
int combinazioni(int n, int k) {
    if (k == 0 || k == n) return 1;
    if (k > n) return 0;

    return combinazioni(n-1, k-1) + combinazioni(n-1, k);
}
```

ALGORITMO ITERATIVO (più efficiente)

```c
int combinazioni_iterativo(int n, int k) {
    if (k > n - k) k = n - k; // Sfrutta simmetria

    long long result = 1;
    for (int i = 0; i < k; i++) {
        result = result * (n - i) / (i + 1);
    }
    return (int)result;
}
```

CATEGORIA 9: FUNZIONI DI SUPPORTO

9.1 FUNZIONI PER LISTE

PRE_INSERT - Inserimento in Testa

```c
void pre_insert(Lista **ptrPtr, int val) {
    Lista *tmp = *ptrPtr;
    *ptrPtr = malloc(sizeof(Lista));
    (*ptrPtr)->valore = val;
    (*ptrPtr)->nextPtr = tmp;
}
```

SUF_INSERT - Inserimento in Coda

```c
void suf_insert(Lista **ptrPtr, int val) {
    while (*ptrPtr != NULL) {
        ptrPtr = &((*ptrPtr)->nextPtr);
    }
    Lista *tmpPtr = *ptrPtr;
    *ptrPtr = malloc(sizeof(Lista));
    (*ptrPtr)->valore = val;
    (*ptrPtr)->nextPtr = tmpPtr;
}
```

PRINT_LIST - Stampa Lista

```c
void print_list(Lista *ptr) {
    if (ptr == NULL) {
        printf("lista vuota\n");
    } else {
        while (ptr != NULL) {
            printf("%d ", ptr->valore);
```

```
            ptr = ptr->nextPtr;
        }
        printf("\n");
    }
}
```

FREE_LIST - Libera Memoria Lista

```
void free_list(Lista *ptr) {
    while (ptr != NULL) {
        Lista *temp = ptr;
        ptr = ptr->nextPtr;
        free(temp);
    }
}
```

9.2 FUNZIONI PER ALBERI

FREE_TREE - Libera Memoria Albero

```
void free_tree(BTree *root) {
    if (root != NULL) {
        free_tree(root->leftPtr);
        free_tree(root->rightPtr);
        free(root);
    }
}
```

COUNT_NODES - Conta Nodi

```
int count_nodes(BTree *root) {
    if (root == NULL) return 0;
    return 1 + count_nodes(root->leftPtr) + count_nodes(root->rightPtr);
}
```

SCHEMA GENERALE PER FUNZIONI RICORSIVE

Template Base

```
TipoRitorno funzione_ricorsiva(ParametriFunzione) {
    // Caso base
    if (condizioneBase) {
        return valoreDiBase;
```

```
    }

    // Caso ricorsivo
    else {
        // Operazioni preliminari
        TipoRitorno risultatoRicorsivo =
funzione_ricorsiva(parametriRidotti);
        // Combinazione del risultato
        return combina(risultatoRicorsivo, altriValori);
    }
}
```

Principi Fondamentali

1. **Caso Base:** Sempre presente e raggiungibile
2. **Progresso:** Ogni chiamata deve avvicinarsi al caso base
3. **Proprietà Invariante:** Mantenuta ad ogni livello
4. **PRE/POST:** Specificate chiaramente per ogni funzione

Gestione Errori

```
#include <assert.h>

void check_preconditions(int condition, char* message) {
    if (!condition) {
        printf("Errore: %s\n", message);
        exit(1);
    }
}
```

Raccolta completa: tutte le funzioni degli esami 2022-2025 con versioni ricorsive e iterative