ESERCIZIO 1: LISTA ORDINATA DECRESCENTE (7 punti, CFU ≥ 6)

CONSEGNA

Scrivere una funzione ricorsiva `inserisci_decrescente` che inserisce un elemento in una lista ordinata in modo decrescente, mantenendo l'ordine. La lista è rappresentata con la seguente struttura:

```
struct nodo {
    int valore;
    struct nodo *next;
};
typedef struct nodo Lista;

void inserisci_decrescente(Lista **head, int val);
```

Richiesto:

1. Scrivere PRE e POST della funzione
2. Implementare la funzione ricorsiva
3. Calcolare e giustificare la complessità temporale

SOLUZIONE

PRE e POST

```
/*
PRE: *head punta ad una lista ordinata in modo decrescente (può essere NULL)
POST: val è stato inserito nella lista mantenendo l'ordine decrescente
*/
```

IMPLEMENTAZIONE RICORSIVA

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

```
    // Caso ricorsivo: continua nella lista
    inserisci_decrescente(&((*head)->next), val);
}
```

ANALISI COMPLESSITÀ

Complessità Temporale: O(n)

- **Caso migliore:** O(1) - inserimento in testa
- **Caso peggiore:** O(n) - inserimento in coda
- **Caso medio:** O(n/2) = O(n)

Giustificazione: Nel caso peggiore si attraversa tutta la lista di n elementi.

Complessità Spaziale: O(n) per lo stack di ricorsione nel caso peggiore.

ESERCIZIO 2: CORRETTEZZA FUNZIONE RICORSIVA (5 punti, CFU ≥ 3)

CONSEGNA

Data la seguente funzione, scrivere PRE e POST condizioni e dimostrarne la correttezza tramite induzione.

```
int contaPositivi(int arr[], int dim) {
    if (dim == 0) {
        return 0;
    }
    if (arr[0] > 0) {
        return 1 + contaPositivi(arr + 1, dim - 1);
    } else {
        return contaPositivi(arr + 1, dim - 1);
    }
}
```

SOLUZIONE

PRE e POST

```
/*
PRE: dim >= 0 e arr contiene almeno dim elementi validi
```

```
POST: restituisce il numero di elementi positivi in arr[0..dim-1]
*/
```

DIMOSTRAZIONE PER INDUZIONE

Proprietà P(n): Per ogni array arr di n elementi, contaPositivi(arr, n) restituisce il numero di elementi positivi.

Caso Base (n = 0):

- contaPositivi(arr, 0) = 0
- In un array vuoto ci sono 0 elementi positivi ✓

Passo Induttivo: Assumiamo P(k) vera per k < n. Dimostriamo P(n):

- Se arr[0] > 0: contaPositivi(arr, n) = 1 + contaPositivi(arr+1, n-1) Per ipotesi induttiva, contaPositivi(arr+1, n-1) conta correttamente i positivi negli elementi 1..n-1 Quindi 1 + contaPositivi(arr+1, n-1) conta correttamente tutti i positivi ✓
- Se arr[0] ≤ 0: contaPositivi(arr, n) = contaPositivi(arr+1, n-1) Per ipotesi induttiva, questo conta correttamente i positivi negli elementi 1..n-1, che sono tutti i positivi ✓

Terminazione: dim decresce ad ogni chiamata e dim ≥ 0, quindi si raggiunge il caso base.

ESERCIZIO 3: ALBERO BINARIO - SOMMA FOGLIE (8 punti, CFU ≥ 6)

CONSEGNA

Implementare una funzione ricorsiva `sommaFoglie` che calcola la somma di tutti i nodi foglia di un albero binario.

```c
struct btree {
    int value;
    struct btree *left;
    struct btree *right;
};
typedef struct btree BTree;

int sommaFoglie(BTree *root);
```

Richiesto:

1. Implementazione ricorsiva
2. Implementazione iterativa (usando stack)

3. Analisi complessità per entrambe

SOLUZIONE

IMPLEMENTAZIONE RICORSIVA

```c
int sommaFoglie(BTree *root) {
    // Caso base: albero vuoto
    if (root == NULL) {
        return 0;
    }

    // Caso base: nodo foglia
    if (root->left == NULL && root->right == NULL) {
        return root->value;
    }

    // Caso ricorsivo: nodo interno
    return sommaFoglie(root->left) + sommaFoglie(root->right);
}
```

IMPLEMENTAZIONE ITERATIVA

```c
#include <stdlib.h>

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

Stack* creaStack(int capacity) {
    Stack *stack = malloc(sizeof(Stack));
    stack->items = malloc(capacity * sizeof(BTree*));
    stack->top = -1;
    stack->capacity = capacity;
    return stack;
}

void push(Stack *stack, BTree *item) {
    if (stack->top < stack->capacity - 1) {
        stack->items[++stack->top] = item;
    }
}

BTree* pop(Stack *stack) {
    if (stack->top >= 0) {
        return stack->items[stack->top--];
```

```
    }
    return NULL;
}

int isEmpty(Stack *stack) {
    return stack->top == -1;
}

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

    Stack *stack = creaStack(1000);
    push(stack, root);
    int somma = 0;

    while (!isEmpty(stack)) {
        BTree *current = pop(stack);

        // Nodo foglia
        if (current->left == NULL && current->right == NULL) {
            somma += current->value;
        } else {
            // Aggiungi figli allo stack
            if (current->right != NULL) {
                push(stack, current->right);
            }
            if (current->left != NULL) {
                push(stack, current->left);
            }
        }
    }

    free(stack->items);
    free(stack);
    return somma;
}
```

ANALISI COMPLESSITÀ

Versione Ricorsiva:

- **Tempo:** O(n) - visita ogni nodo una volta
- **Spazio:** O(h) dove h è l'altezza dell'albero (stack di ricorsione)

Versione Iterativa:

- **Tempo:** O(n) - visita ogni nodo una volta
- **Spazio:** O(w) dove w è la larghezza massima dell'albero

ESERCIZIO 4: MATRICE - PERCORSO MASSIMO (9 punti, CFU ≥ 9)

CONSEGNA

Implementare una funzione che trova il percorso con somma massima da (0,0) a (n-1, m-1) in una matrice. Si può muovere solo verso destra o verso il basso.

```
int percorsoMassimo(int matrice[MAX_RIGHE][MAX_COLONNE], int righe, int
colonne);
```

Richiesto:

1. Soluzione ricorsiva
2. Soluzione con programmazione dinamica
3. Confronto delle complessità

SOLUZIONE

SOLUZIONE RICORSIVA (NAIVE)

```
#define MAX_RIGHE 100
#define MAX_COLONNE 100

int percorsoMassimoRec(int matrice[MAX_RIGHE][MAX_COLONNE], int i, int j,
int righe, int colonne) {
    // Caso base: destinazione raggiunta
    if (i == righe - 1 && j == colonne - 1) {
        return matrice[i][j];
    }

    // Fuori dai limiti
    if (i >= righe || j >= colonne) {
        return INT_MIN; // Valore impossibile
    }

    // Ricorsione: scegli il massimo tra destra e giù
    int destra = percorsoMassimoRec(matrice, i, j + 1, righe, colonne);
    int giu = percorsoMassimoRec(matrice, i + 1, j, righe, colonne);

    return matrice[i][j] + ((destra > giu) ? destra : giu);
}

int percorsoMassimo(int matrice[MAX_RIGHE][MAX_COLONNE], int righe, int
colonne) {
```

```
    return percorsoMassimoRec(matrice, 0, 0, righe, colonne);
}
```

SOLUZIONE CON PROGRAMMAZIONE DINAMICA

```
int percorsoMassimoDP(int matrice[MAX_RIGHE][MAX_COLONNE], int righe, int
colonne) {
    int dp[MAX_RIGHE][MAX_COLONNE];

    // Inizializza prima cella
    dp[0][0] = matrice[0][0];

    // Inizializza prima riga
    for (int j = 1; j < colonne; j++) {
        dp[0][j] = dp[0][j-1] + matrice[0][j];
    }

    // Inizializza prima colonna
    for (int i = 1; i < righe; i++) {
        dp[i][0] = dp[i-1][0] + matrice[i][0];
    }

    // Riempi la tabella
    for (int i = 1; i < righe; i++) {
        for (int j = 1; j < colonne; j++) {
            int dalAlto = dp[i-1][j];
            int daSinistra = dp[i][j-1];
            dp[i][j] = matrice[i][j] + ((dalAlto > daSinistra) ? dalAlto :
daSinistra);
        }
    }

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

CONFRONTO COMPLESSITÀ

Ricorsiva Naive:

- **Tempo:** O(2^(n+m)) - esponenziale
- **Spazio:** O(n+m) - profondità ricorsione

Programmazione Dinamica:

- **Tempo:** O(n×m) - polinomiale
- **Spazio:** O(n×m) - per la tabella DP

Ottimizzazione Spazio DP:

```c
int percorsoMassimoOptimized(int matrice[MAX_RIGHE][MAX_COLONNE], int righe,
int colonne) {
    int dp[MAX_COLONNE];

    // Inizializza prima riga
    dp[0] = matrice[0][0];
    for (int j = 1; j < colonne; j++) {
        dp[j] = dp[j-1] + matrice[0][j];
    }

    // Processa riga per riga
    for (int i = 1; i < righe; i++) {
        dp[0] += matrice[i][0]; // Prima colonna
        for (int j = 1; j < colonne; j++) {
            dp[j] = matrice[i][j] + ((dp[j] > dp[j-1]) ? dp[j] : dp[j-1]);
        }
    }

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

Spazio ottimizzato: O(m) invece di O(n×m)

ESERCIZIO 5: DANGLING POINTERS E GESTIONE MEMORIA (6 punti, CFU ≥ 3)

CONSEGNA

Analizzare il seguente codice, identificare tutti i problemi di gestione memoria e correggerli:

```c
Lista* creaLista() {
    Lista lista[3];
    lista[0].valore = 1; lista[0].next = &lista[1];
    lista[1].valore = 2; lista[1].next = &lista[2];
    lista[2].valore = 3; lista[2].next = NULL;
    return lista; // PROBLEMA 1
}

void aggiungiElemento(Lista *head, int val) {
    Lista *nuovo = malloc(sizeof(Lista));
    nuovo->valore = val;
    nuovo->next = head;
    head = nuovo; // PROBLEMA 2
}
```

```c
void stampaLista(Lista *head) {
    Lista *temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        printf("%d ", temp->valore);
        free(temp); // PROBLEMA 3
    }
}
```

SOLUZIONE

PROBLEMI IDENTIFICATI

PROBLEMA 1: Dangling Pointer

- `lista` è un array locale che viene deallocato al termine della funzione
- Il puntatore restituito punta a memoria non valida

PROBLEMA 2: Modifica Parametro per Valore

- `head` è passato per valore, le modifiche non si riflettono nel chiamante
- Serve passare `Lista **head`

PROBLEMA 3: Distruzione durante Iterazione

- La funzione `stampaLista` non dovrebbe liberare la memoria
- Dovrebbe esistere una funzione separata per deallocare

CODICE CORRETTO

```c
Lista* creaLista() {
    // Alloca in heap invece che in stack
    Lista *primo = malloc(sizeof(Lista));
    Lista *secondo = malloc(sizeof(Lista));
    Lista *terzo = malloc(sizeof(Lista));

    if (!primo || !secondo || !terzo) {
        free(primo); free(secondo); free(terzo);
        return NULL;
    }

    primo->valore = 1; primo->next = secondo;
    secondo->valore = 2; secondo->next = terzo;
    terzo->valore = 3; terzo->next = NULL;

    return primo;
}
```

```c
void aggiungiElemento(Lista **head, int val) {
    Lista *nuovo = malloc(sizeof(Lista));
    if (nuovo == NULL) {
        printf("Errore allocazione memoria\n");
        return;
    }

    nuovo->valore = val;
    nuovo->next = *head;
    *head = nuovo;
}

void stampaLista(Lista *head) {
    while (head != NULL) {
        printf("%d ", head->valore);
        head = head->next;
    }
    printf("\n");
}

void liberaLista(Lista *head) {
    Lista *temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

// Uso corretto:
int main() {
    Lista *lista = creaLista();
    aggiungiElemento(&lista, 0);
    stampaLista(lista);
    liberaLista(lista);
    return 0;
}
```

ESERCIZIO 6: COMPLESSITÀ E OTTIMIZZAZIONE (7 punti, CFU ≥ 6)

CONSEGNA

Analizzare la complessità delle seguenti funzioni e proporre versioni ottimizzate quando possibile:

```java
// Funzione A
int funzioneA(int n) {
    int somma = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            for (int k = 1; k <= j; k++) {
                somma++;
            }
        }
    }
    return somma;
}

// Funzione B
int funzioneB(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            if (arr[i] + arr[j] == target) {
                return 1;
            }
        }
    }
    return 0;
}
```

SOLUZIONE

ANALISI FUNZIONE A

Complessità:

$$
\begin{aligned}
\text{Somma} &= \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} 1 \\
&= \sum_{i=1}^{n} \sum_{j=1}^{i} j \\
&= \sum_{i=1}^{n} \frac{i(i+1)}{2} \\
&= \frac{1}{2} \sum_{i=1}^{n} (i^2 + i) \\
&= \frac{1}{2}\left[\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}\right] \\
&= O(n^3)
\end{aligned}
$$

Versione Ottimizzata:

```java
int funzioneA_ottimizzata(int n) {
    // Formula chiusa: calcolo diretto in O(1)
    long long result = (long long)n * (n + 1) * (n + 2) / 6;
    return (int)result;
}
```

ANALISI FUNZIONE B

Complessità: O(n²) - due cicli innestati

Versione Ottimizzata (con hash table):

```c
#include <stdbool.h>
#define MAX_VAL 10000

int funzioneB_ottimizzata(int arr[], int n, int target) {
    bool visto[2 * MAX_VAL + 1] = {false}; // Assuming values in [-MAX_VAL, MAX_VAL]

    for (int i = 0; i < n; i++) {
        int complemento = target - arr[i];

        // Controlla se il complemento è già stato visto
        if (complemento >= -MAX_VAL && complemento <= MAX_VAL) {
            if (visto[complemento + MAX_VAL]) {
                return 1;
            }
        }

        // Marca il valore corrente come visto
        if (arr[i] >= -MAX_VAL && arr[i] <= MAX_VAL) {
            visto[arr[i] + MAX_VAL] = true;
        }
    }

    return 0;
}
```

Complessità ottimizzata: O(n) tempo, O(1) spazio (assumendo range limitato)

CONFRONTO COMPLESSITÀ

Funzione	Versione Originale	Versione Ottimizzata	Miglioramento
A	O(n³)	O(1)	Enorme
B	O(n²)	O(n)	Quadratico

ESERCIZIO 7: ARITMETICA PUNTATORI AVANZATA (5 punti, CFU ≥ 3)

CONSEGNA

Data la seguente dichiarazione e inizializzazione:

```
int matrice[4][3][2] = {
    {{1,2}, {3,4}, {5,6}},
    {{7,8}, {9,10}, {11,12}},
    {{13,14}, {15,16}, {17,18}},
    {{19,20}, {21,22}, {23,24}}
};
int *p = (int*)matrice;
```

Rispondere alle seguenti domande:

1. Qual è il valore di `*(p + 7)`?
2. Come accedere a `matrice[2][1][0]` usando solo aritmetica dei puntatori?
3. Se `matrice[i][j][k]` è in posizione `pos` nell'array lineare, scrivere la formula per calcolare `i`, `j`, `k` dato `pos`.

SOLUZIONE

1. Valore di *(p + 7)

La matrice è memorizzata linearmente come:

```
pos: 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
val: 1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

Risposta: `*(p + 7)` = **8**

2. Accesso a matrice[2][1][0]

Formula generale per matrice[i][j][k] con dimensioni [4][3][2]:

```
posizione = i * (3 * 2) + j * 2 + k
```

Per `matrice[2][1][0]`:

```
posizione = 2 * 6 + 1 * 2 + 0 = 12 + 2 + 0 = 14
```

Risposta: `*(p + 14)`

3. Formula inversa

Date le dimensioni [4][3][2] e posizione `pos`:

```
int k = pos % 2;
int j = (pos / 2) % 3;
int i = pos / (3 * 2);

// Verifica: pos = i * 6 + j * 2 + k
```

Formule generali per matrice[D1][D2][D3]:

```
k = pos % D3;
j = (pos / D3) % D2;
i = pos / (D2 * D3);
```

ESERCIZIO 8: RICORSIONE CON MEMOIZATION (8 punti, CFU ≥ 9)

CONSEGNA

Il problema del "Taglio delle Aste": data un'asta di lunghezza n e un array di prezzi prices[i] per aste di lunghezza i+1, trovare il guadagno massimo tagliando l'asta.

```
int taglioAste(int n, int prices[]);
```

Implementare:

1. Versione ricorsiva naive
2. Versione con memoization
3. Versione bottom-up
4. Analisi complessità di tutte e tre

SOLUZIONE

1. VERSIONE RICORSIVA NAIVE

```
int taglioAsteNaive(int n, int prices[]) {
    // Caso base
    if (n == 0) return 0;

    int maxVal = 0;

    // Prova tutti i possibili tagli
```

```
    for (int i = 1; i <= n; i++) {
        int val = prices[i-1] + taglioAsteNaive(n - i, prices);
        if (val > maxVal) {
            maxVal = val;
        }
    }

    return maxVal;
}
```

Complessità: O(2^n) - esponenziale

2. VERSIONE CON MEMOIZATION (TOP-DOWN)

```
#include <string.h>

int taglioAsteMemoHelper(int n, int prices[], int memo[]) {
    // Controlla se già calcolato
    if (memo[n] != -1) {
        return memo[n];
    }

    // Caso base
    if (n == 0) {
        return memo[n] = 0;
    }

    int maxVal = 0;

    // Prova tutti i possibili tagli
    for (int i = 1; i <= n; i++) {
        int val = prices[i-1] + taglioAsteMemoHelper(n - i, prices, memo);
        if (val > maxVal) {
            maxVal = val;
        }
    }

    return memo[n] = maxVal;
}

int taglioAsteMemo(int n, int prices[]) {
    int *memo = malloc((n + 1) * sizeof(int));

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

    int result = taglioAsteMemoHelper(n, prices, memo);
```

```
    free(memo);

    return result;
}
```

Complessità: O(n²) tempo, O(n) spazio

3. VERSIONE BOTTOM-UP (DYNAMIC PROGRAMMING)

```
int taglioAsteDP(int n, int prices[]) {
    int *dp = calloc(n + 1, sizeof(int));

    // dp[0] = 0 (già inizializzato da calloc)

    // Riempi la tabella dal basso verso l'alto
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            int val = prices[j-1] + dp[i-j];
            if (val > dp[i]) {
                dp[i] = val;
            }
        }
    }

    int result = dp[n];
    free(dp);

    return result;
}
```

Complessità: O(n²) tempo, O(n) spazio

4. CONFRONTO COMPLESSITÀ

Approccio	Tempo	Spazio	Note
Naive	O(2^n)	O(n)	Stack ricorsione
Memoization	O(n²)	O(n)	Stack + tabella
Bottom-up	O(n²)	O(n)	Solo tabella

ESEMPIO DI ESECUZIONE

```
int main() {
    int prices[] = {1, 5, 8, 9, 10, 17, 17, 20};
    int n = 8;
```

```c
    printf("Naive: %d\n", taglioAsteNaive(n, prices));        // Lento per n
grande
    printf("Memo: %d\n", taglioAsteMemo(n, prices));          // Veloce
    printf("DP: %d\n", taglioAsteDP(n, prices));              // Veloce

    return 0;
}
```

RIEPILOGO COMPLESSITÀ DEGLI ESERCIZI

Esercizio	Algoritmo	Complessità Tempo	Complessità Spazio
1	Lista Ordinata	O(n)	O(n) ricorsione
2	Conta Positivi	O(n)	O(n) ricorsione
3	Somma Foglie	O(n)	O(h) altezza albero
4	Percorso Massimo (DP)	O(n×m)	O(m) ottimizzato
5	Gestione Memoria	-	-
6	Ottimizzazioni	O(1), O(n)	O(1)
7	Aritmetica Puntatori	-	-
8	Taglio Aste (DP)	$O(n^2)$	O(n)

Esercizi simulati basati sui pattern reali degli esami 2022-2025