CATEGORIA 1: COSA STAMPA - ARITMETICA PUNTATORI AVANZATA

Quiz 1.1 - Matrici e Puntatori Complessi

```
int matrix[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
int *p = (int*)matrix;
int **q = (int**)matrix;

printf("%d %d %d\n", *(p+5), *(*q+2), *(*(matrix+1)+2));
```

Opzioni:

1. `6 3 7`
2. `6 2 8`
3. `5 3 7`
4. Errore di compilazione

Risposta: 1. `6 3 7`

Spiegazione:

- `*(p+5)` : p punta a matrix[0][0], p+5 punta al 6° elemento (matrix[1][1]) = 6
- `*(*q+2)` : q interpretato come int*, q è il primo "puntatore", *q+2 punta al 3° int = 3
- `*(*(matrix+1)+2)` : matrix+1 punta alla 2° riga, *(matrix+1)+2 punta al 3° elemento della 2° riga = 7

Quiz 1.2 - Dangling Pointers e Scope

```
int* funzione() {
    int arr[3] = {10, 20, 30};
    int *p = arr;
    return p + 1;
}

int main() {
    int *ptr = funzione();
    printf("%d", *ptr);
    return 0;
}
```

Opzioni:

1. `20`
2. `10`
3. Comportamento indefinito
4. Errore di compilazione

Risposta: 3. Comportamento indefinito

Spiegazione: `arr` è un array locale che viene deallocato quando `funzione()` termina. Il puntatore restituito è un dangling pointer.

Quiz 1.3 - Aritmetica con Cast e Incrementi

```
char str[] = "HELLO";
char *p = str;
int *ip = (int*)p;

printf("%d %c %d\n", (int)*p, *(p+2), (int)*(p+4));
```

Assumendo ASCII: H=72, E=69, L=76, O=79

Opzioni:

1. `72 L 79`
2. `72 L 0`
3. `72 L O`
4. Comportamento indefinito

Risposta: 2. `72 L 0`

Spiegazione:

- `(int)*p` : converte 'H' (72) in int = 72
- `*(p+2)` : carattere in posizione 2 = 'L'
- `(int)*(p+4)` : carattere in posizione 4 = '\0' (terminatore) = 0

Quiz 1.4 - Puntatori a Strutture

```
struct punto {
    int x, y;
};

struct punto arr[2] = {{1,2}, {3,4}};
struct punto *p = arr;
int *ip = (int*)p;
```

```
printf("%d %d %d\n", p->x, (p+1)->y, *(ip+3));
```

Opzioni:

1. `1 4 4`
2. `1 4 3`
3. `1 2 4`
4. Errore di compilazione

Risposta: 1. `1 4 4`

Spiegazione:

- `p->x` : x del primo punto = 1
- `(p+1)->y` : y del secondo punto = 4
- `*(ip+3)` : ip punta agli int della struttura: [1,2,3,4], ip+3 punta al 4° = 4

CATEGORIA 2: COMPLESSITÀ COMPUTAZIONALE

Quiz 2.1 - Analisi Cicli Innestati

```
void funzione(int n) {
    for (int i = 1; i <= n; i *= 2) {
        for (int j = 1; j <= i; j++) {
            for (int k = 1; k <= n; k += j) {
                // Operazione O(1)
            }
        }
    }
}
```

Qual è la complessità temporale?

Opzioni:

1. $O(n^2)$
2. $O(n^2 \log n)$
3. $O(n^3)$
4. $O(n \log^2 n)$

Risposta: 2. $O(n^2 \log n)$

Spiegazione:

- Ciclo esterno: i = 1,2,4,8,...,n → O(log n) iterazioni
- Per ogni i: ciclo j va da 1 a i → O(i) iterazioni
- Per ogni j: ciclo k fa n/j iterazioni → O(n/j)
- Totale: Σ(i=2^0 to 2^log n) Σ(j=1 to i) n/j ≈ O(n log n · log n) = O(n log² n)

Quiz 2.2 - Ricorsione con Branching

```
int mystery(int n) {
    if (n <= 1) return 1;
    return mystery(n/3) + mystery(n/3) + mystery(n/3);
}
```

Qual è la complessità temporale?

Opzioni:

1. O(n)
2. O(3^(log₃ n))
3. O(n^log₃ 3)
4. O(log n)

Risposta: 3. O(n^log₃ 3) = O(n)

Spiegazione:

- Master Theorem: T(n) = 3T(n/3) + O(1)
- a=3, b=3, f(n)=O(1)
- log_b(a) = log₃(3) = 1
- f(n) = O(n^0) = O(1) = O(n^(1-ε)) per ε>0
- Caso 1: T(n) = O(n^log₃ 3) = O(n¹) = O(n)

Quiz 2.3 - Spazio vs Tempo

```
// Versione A
int fibA(int n) {
    if (n <= 1) return n;
    return fibA(n-1) + fibA(n-2);
}

// Versione B
int fibB(int n) {
    int a = 0, b = 1, temp;
    for (int i = 2; i <= n; i++) {
        temp = a + b;
        a = b;
```

```
        b = temp;
    }
    return (n == 0) ? 0 : b;
}
```

Confronto delle complessità:

Opzioni:

1. A: O(2ⁿ) tempo, O(n) spazio; B: O(n) tempo, O(1) spazio
2. A: O(n²) tempo, O(n) spazio; B: O(n) tempo, O(n) spazio
3. A: O(2ⁿ) tempo, O(2ⁿ) spazio; B: O(n) tempo, O(1) spazio
4. Entrambe O(n) tempo, O(1) spazio

Risposta: 1. A: O(2ⁿ) tempo, O(n) spazio; B: O(n) tempo, O(1) spazio

Spiegazione:

- Versione A: ricorsione esponenziale con stack depth O(n)
- Versione B: iterazione lineare con spazio costante

CATEGORIA 3: CORRETTEZZA E DIMOSTRAZIONE

Quiz 3.1 - Invariante di Ciclo

```
int ricerca(int arr[], int n, int target) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

Qual è l'invariante di ciclo corretto?

Opzioni:

1. `arr[low] ≤ target ≤ arr[high]`
2. Se target è presente, è nell'intervallo [low, high]
3. `low ≤ mid ≤ high`
4. `arr[mid] ≠ target`

Risposta: 2. Se target è presente, è nell'intervallo [low, high]

Spiegazione: L'invariante mantiene che se l'elemento target esiste nell'array, deve trovarsi nella porzione corrente [low, high].

Quiz 3.2 - PRE/POST Conditions

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

Quale coppia PRE/POST è corretta?

Opzioni:

1. PRE: a ≠ NULL, b ≠ NULL; POST: *a e* b sono scambiati
2. PRE: a > 0, b > 0; POST: a = old_b, b = old_a
3. PRE: a e b puntano a interi validi; POST: contenuto puntato scambiato
4. PRE: nessuna; POST: a = b, b = a

Risposta: 3. PRE: a e b puntano a interi validi; POST: contenuto puntato scambiato

Spiegazione: La precondizione deve garantire che i puntatori siano validi, la postcondizione descrive l'effetto dello scambio.

Quiz 3.3 - Terminazione Ricorsiva

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

Perché la ricorsione termina sempre (assumendo a,b > 0)?

Opzioni:

1. Perché `a % b < b` sempre
2. Perché `b` decresce strettamente ad ogni chiamata
3. Perché `a + b` decresce
4. Perché raggiunge sempre `b = 0`

Risposta: 1. Perché `a % b < b` sempre

Spiegazione: Ad ogni chiamata, il secondo argomento diventa `a % b` che è strettamente minore di `b`, garantendo la convergenza verso 0.

CATEGORIA 4: DICHIARAZIONI COMPLESSE AVANZATE

Quiz 4.1 - Puntatori a Funzioni

Interpretare: `int (*(*fp)[10])(int, char*);`

Opzioni:

1. fp è un puntatore a un array di 10 puntatori a funzioni
2. fp è un array di 10 puntatori a funzioni
3. fp è un puntatore a una funzione che restituisce un array
4. fp è una funzione che restituisce un puntatore

Risposta: 1. fp è un puntatore a un array di 10 puntatori a funzioni

Spiegazione:

- `(*fp)[10]` : fp è puntatore a array di 10 elementi
- `int (*)(int, char*)` : ogni elemento è puntatore a funzione

Quiz 4.2 - Precedenza Operatori

Data: `int *p[5][3];`

Cosa rappresenta?

Opzioni:

1. p è un puntatore a matrice 5×3 di interi
2. p è una matrice 5×3 di puntatori a intero
3. p è un array di 5 puntatori a array di 3 interi
4. Dichiarazione non valida

Risposta: 2. p è una matrice 5×3 di puntatori a intero

Spiegazione: `[]` ha precedenza maggiore di `*`, quindi si legge come `int *(p[5][3])`.

Quiz 4.3 - Const e Puntatori

Analizzare: `const int * const * const p;`

Opzioni:

1. p è costante, punta a puntatore costante a int costante
2. Solo p è costante
3. Solo l'int è costante
4. Tutto è costante

Risposta: 1. p è costante, punta a puntatore costante a int costante

Spiegazione:

- `const int`: l'intero è costante
- `* const`: il primo puntatore è costante
- `* const`: p (secondo puntatore) è costante

CATEGORIA 5: GESTIONE MEMORIA E ERRORI

Quiz 5.1 - Memory Leak Detection

```
void funzione() {
    int *p = malloc(100 * sizeof(int));
    int *q = malloc(50 * sizeof(int));

    if (some_condition) {
        free(p);
        return;  // Punto A
    }

    free(p);
    free(q);  // Punto B
}
```

Quale affermazione è corretta?

Opzioni:

1. Memory leak sempre al punto A
2. Memory leak sempre al punto B
3. Memory leak solo se some_condition è vero
4. Memory leak solo se some_condition è falso

Risposta: 3. Memory leak solo se some_condition è vero

Spiegazione: Se some_condition è vero, si esce al punto A senza liberare q, causando memory leak.

Quiz 5.2 - Double Free

```c
void problema() {
    int *p = malloc(sizeof(int));
    int *q = p;

    free(p);
    free(q);  // Problema qui
}
```

Quale errore si verifica?

Opzioni:

1. Memory leak
2. Double free
3. Dangling pointer dereference
4. Segmentation fault (ma non double free)

Risposta: 2. Double free

Spiegazione: p e q puntano alla stessa memoria, liberarla due volte è un double free error.

Quiz 5.3 - Use After Free

```c
int* crea_array(int n) {
    int *arr = malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        arr[i] = i * i;
    }
    free(arr);
    return arr;  // Problema
}
```

Quale pattern di errore è questo?

Opzioni:

1. Memory leak
2. Use after free
3. Return di puntatore a variabile locale
4. Buffer overflow

Risposta: 2. Use after free

Spiegazione: La funzione libera la memoria ma restituisce il puntatore alla memoria liberata.

CATEGORIA 6: LOGICA BOOLEANA COMPLESSA

Quiz 6.1 - Short-Circuit e Side Effects

```c
int x = 5, y = 10;
if ((x++ > 5) && (y++ > 10)) {
    printf("Vero\n");
} else {
    printf("Falso\n");
}
printf("%d %d\n", x, y);
```

Cosa stampa?

Opzioni:

1. `Falso\n6 11`
2. `Falso\n6 10`
3. `Vero\n6 11`
4. `Falso\n5 10`

Risposta: 2. `Falso\n6 10`

Spiegazione:

- `x++ > 5` : x diventa 6, ma 5 > 5 è falso
- Short-circuit: `y++` non viene eseguito
- Risultato: Falso, x=6, y=10

Quiz 6.2 - Operatori Bitwise vs Logici

```c
int a = 3, b = 6;  // a=011, b=110 in binario
printf("%d %d %d\n", a && b, a & b, a || b);
```

Cosa stampa?

Opzioni:

1. `1 2 1`

2. `0 2 1`
3. `1 3 1`
4. `3 2 6`

Risposta: 1. `1 2 1`

Spiegazione:

- `a && b`: logico AND → entrambi non-zero → 1
- `a & b`: bitwise AND → 011 & 110 = 010 = 2
- `a || b`: logico OR → almeno uno non-zero → 1

Quiz 6.3 - Precedenza e Associatività

```c
int x = 2, y = 3, z = 4;
int result = x < y < z;
printf("%d\n", result);
```

Cosa stampa?

Opzioni:

1. `0`
2. `1`
3. `4`
4. Errore di compilazione

Risposta: 2. `1`

Spiegazione:

- Associatività sinistra: `(x < y) < z`
- `(2 < 3) < 4` → `1 < 4` → `1`

CATEGORIA 7: RICORSIONE E PATTERN ALGORITMICI

Quiz 7.1 - Tail Recursion

```c
int fattoriale(int n, int acc) {
    if (n <= 1) return acc;
```

```
    return fattoriale(n - 1, n * acc);
}
```

Quale affermazione è corretta?

Opzioni:

1. Non è tail recursive
2. È tail recursive e ottimizzabile
3. Ha complessità spaziale O(n) sempre
4. Non calcola il fattoriale correttamente

Risposta: 2. È tail recursive e ottimizzabile

Spiegazione: La chiamata ricorsiva è l'ultima operazione, permettendo l'ottimizzazione tail call.

Quiz 7.2 - Backtracking Pattern

```
int trova_percorso(int labirinto[N][N], int x, int y, int sol[N][N]) {
    if (x == N-1 && y == N-1) {
        sol[x][y] = 1;
        return 1;
    }

    if (valido(x, y, labirinto)) {
        sol[x][y] = 1;

        if (trova_percorso(labirinto, x+1, y, sol) ||
            trova_percorso(labirinto, x, y+1, sol)) {
            return 1;
        }

        sol[x][y] = 0;  // Backtrack
        return 0;
    }

    return 0;
}
```

Qual è lo scopo di `sol[x][y] = 0`?

Opzioni:

1. Inizializzazione
2. Backtracking - annulla la scelta se non porta a soluzione
3. Marcatura di celle visitate

4. Prevenzione di cicli infiniti

Risposta: 2. Backtracking - annulla la scelta se non porta a soluzione

Spiegazione: Il backtracking rimuove la marca quando un percorso non porta alla soluzione.

Quiz 7.3 - Divide et Impera

```
int max_subarray(int arr[], int low, int high) {
    if (low == high) return arr[low];

    int mid = (low + high) / 2;
    int left_max = max_subarray(arr, low, mid);
    int right_max = max_subarray(arr, mid+1, high);
    int cross_max = max_crossing_sum(arr, low, mid, high);

    return max(left_max, max(right_max, cross_max));
}
```

Quale problema risolve questo algoritmo?

Opzioni:

1. Ricerca binaria
2. Maximum subarray sum (Kadane's algorithm variante)
3. Merge sort
4. Quick select

Risposta: 2. Maximum subarray sum (Kadane's algorithm variante)

Spiegazione: Divide et impera per trovare la sottosequenza contigua con somma massima.

CATEGORIA 8: PATTERN MATCHING E STRINGHE

Quiz 8.1 - Algoritmo KMP

```
void compute_lps(char pattern[], int M, int lps[]) {
    int len = 0, i = 1;
    lps[0] = 0;

    while (i < M) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
```

```
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

Per pattern "ABCABCAB", l'array lps sarà:

Opzioni:

1. `[0,0,0,1,2,3,4,5]`
2. `[0,0,0,1,2,3,1,2]`
3. `[0,1,2,0,1,2,3,4]`
4. `[0,0,0,1,2,3,4,2]`

Risposta: 2. `[0,0,0,1,2,3,1,2]`

Spiegazione:

- A B C A B C A B
- 0 0 0 1 2 3 1 2
- Ogni valore rappresenta la lunghezza del prefisso proprio più lungo che è anche suffisso.

Quiz 8.2 - String Manipulation

```
char* string_reverse(char *str) {
    int n = strlen(str);
    for (int i = 0; i < n/2; i++) {
        char temp = str[i];
        str[i] = str[n-1-i];
        str[n-1-i] = temp;
    }
    return str;
}
```

Qual è la complessità e correttezza?

Opzioni:

1. O(n²) tempo, modifica stringa originale
2. O(n) tempo, modifica stringa originale

3. O(n) tempo, crea nuova stringa
4. O(log n) tempo, modifica stringa originale

Risposta: 2. O(n) tempo, modifica stringa originale

Spiegazione: Fa n/2 scambi, ognuno O(1), modificando la stringa in-place.

CATEGORIA 9: STRUTTURE DATI AVANZATE

Quiz 9.1 - Hash Table

```c
int hash_function(int key, int table_size) {
    return ((key % table_size) + table_size) % table_size;
}
```

Perché si usa `((key % table_size) + table_size) % table_size`?

Opzioni:

1. Per evitare collisioni
2. Per gestire chiavi negative
3. Per migliorare la distribuzione
4. Per aumentare la velocità

Risposta: 2. Per gestire chiavi negative

Spiegazione: In C, il modulo di un numero negativo può essere negativo. L'aggiunta di table_size garantisce un risultato positivo.

Quiz 9.2 - Binary Search Tree

```c
BSTNode* delete_node(BSTNode* root, int key) {
    if (root == NULL) return root;

    if (key < root->data) {
        root->left = delete_node(root->left, key);
    } else if (key > root->data) {
        root->right = delete_node(root->right, key);
    } else {
        // Nodo da eliminare trovato
        if (root->left == NULL) {
            BSTNode* temp = root->right;
            free(root);
            return temp;
```

```
        } else if (root->right == NULL) {
            BSTNode* temp = root->left;
            free(root);
            return temp;
        }

        // Nodo con due figli
        BSTNode* temp = find_min(root->right);
        root->data = temp->data;
        root->right = delete_node(root->right, temp->data);
    }
    return root;
}
```

Qual è la complessità nel caso peggiore?

Opzioni:

1. O(log n) sempre
2. O(n) se albero sbilanciato
3. O(1) se nodo è foglia
4. O(n log n) sempre

Risposta: 2. O(n) se albero sbilanciato

Spiegazione: In un BST sbilanciato (lista), la ricerca può richiedere O(n) operazioni.

CATEGORIA 10: OTTIMIZZAZIONE E PERFORMANCE

Quiz 10.1 - Cache Efficiency

```
// Versione A
void sum_matrix_A(int matrix[1000][1000]) {
    int sum = 0;
    for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < 1000; j++) {
            sum += matrix[i][j];
        }
    }
}

// Versione B
void sum_matrix_B(int matrix[1000][1000]) {
    int sum = 0;
    for (int j = 0; j < 1000; j++) {
```

```
        for (int i = 0; i < 1000; i++) {
            sum += matrix[i][j];
        }
    }
}
```

Quale versione è più efficiente per la cache?

Opzioni:

1. Versione A - accesso row-major (per righe)
2. Versione B - accesso column-major (per colonne)
3. Equivalenti
4. Dipende dalla CPU

Risposta: 1. Versione A - accesso row-major (per righe)

Spiegazione: In C, le matrici sono memorizzate per righe. L'accesso sequenziale per righe sfrutta meglio la locality spaziale della cache.

Quiz 10.2 - Loop Unrolling

```
// Versione normale
for (int i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
}

// Versione unrolled
for (int i = 0; i < n-3; i += 4) {
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
}
```

Quale vantaggio offre il loop unrolling?

Opzioni:

1. Riduce il numero di confronti e salti
2. Usa meno memoria
3. È più leggibile
4. Previene overflow

Risposta: 1. Riduce il numero di confronti e salti

Spiegazione: Il loop unrolling riduce l'overhead del controllo del ciclo, eseguendo più operazioni per iterazione.

RIEPILOGO DIFFICOLTÀ E PUNTEGGI

Distribuzione per Categoria

Categoria	Difficoltà	Punti Tipici	Concetti Chiave
1. Cosa Stampa	Media-Alta	2-3	Puntatori, Cast, Scope
2. Complessità	Alta	3-4	Big-O, Master Theorem
3. Correttezza	Alta	4-5	Invarianti, PRE/POST
4. Dichiarazioni	Media	2-3	Precedenza, Semantica
5. Memoria	Media-Alta	3-4	Leaks, Dangling, Double-free
6. Logica	Media	2-3	Short-circuit, Bitwise
7. Ricorsione	Alta	4-5	Tail recursion, Backtracking
8. Stringhe	Media	3-4	Algoritmi, Pattern matching
9. Strutture Dati	Alta	4-5	BST, Hash, Complessità
10. Ottimizzazione	Molto Alta	5-6	Cache, Performance

Strategie di Risoluzione Rapida

Per Quiz di Complessità:

1. **Identificare i cicli** e loro relazioni
2. **Applicare Master Theorem** se possibile
3. **Contare operazioni** nel caso peggiore
4. **Considerare spazio vs tempo**

Per Quiz "Cosa Stampa":

1. **Tracciare passo-passo** l'esecuzione
2. **Attenzione ai cast** di tipo
3. **Verificare validità** dei puntatori
4. **Considerare side effects** degli operatori

Per Quiz di Correttezza:

1. **Identificare invarianti** del ciclo

2. **Scrivere PRE/POST** esplicite
3. **Verificare terminazione**
4. **Usare induzione** per dimostrazioni

Quiz simulati basati sui pattern reali degli esami 2022-2025 con focus su argomenti critici