Certamente! Ecco le soluzioni per tutti gli esercizi proposti:

1. Attraversamento di un campo fiorito

```c
#define DIM_X 5
#define DIM_Y 5

int attraversa_campo(int campo[DIM_X][DIM_Y], int pos_x, int pos_y) {
    // Caso base: fuori dai limiti o su un fiore
    if (pos_x < 0 || pos_x >= DIM_X || pos_y < 0 || pos_y >= DIM_Y ||
campo[pos_x][pos_y] == 0) {
        return 0;
    }

    // Caso base: raggiunta la cima
    if (pos_x == 0) {
        return 1;
    }

    // Ricorsione: somma dei percorsi andando in alto e a destra
    return attraversa_campo(campo, pos_x - 1, pos_y) + attraversa_campo(campo,
pos_x, pos_y + 1);
}
```

2. Rimozione di triple consecutive

```c
void rimuovi_triple(int *A, int *dim) {
    if (*dim <= 2) return;  // Non ci sono abbastanza elementi per formare una
tripla

    int i, j;
    for (i = 1; i < *dim - 1; i++) {
        if (A[i-1] == A[i] && A[i] == A[i+1]) {
            // Rimuovi l'elemento centrale spostando gli elementi successivi
            for (j = i; j < *dim - 1; j++) {
                A[j] = A[j+1];
            }
            (*dim)--;
            i--;  // Ricontrolla la stessa posizione
        }
    }
}
```

3. Ricerca in array con elementi ordinati a coppie

```
int ricerca_ordinata(int *A, int *B, int dim_A, int dim_B) {
    // Caso base: A è vuoto, quindi tutti gli elementi sono stati trovati
    if (dim_A == 0) return 1;

    // Caso base: B è vuoto ma A no, quindi non tutti gli elementi di A sono
in B
    if (dim_B == 0) return 0;

    // Se il primo elemento di A corrisponde al primo di B (o al suo opposto)
    if (A[0] == B[0] || A[0] == -B[0]) {
        return ricerca_ordinata(A + 1, B + 1, dim_A - 1, dim_B - 1);
    }

    // Altrimenti, continua la ricerca nel resto di B
    return ricerca_ordinata(A, B + 1, dim_A, dim_B - 1);
}
```

4. Clonazione di una lista con skip

```
void clone_list_skip(Lista *srcPtr, Lista **destPtr) {
    *destPtr = NULL;
    Lista *current = srcPtr;
    Lista **tail = destPtr;
    int count = 0;

    while (current != NULL) {
        if (count % 2 == 0) {
            *tail = malloc(sizeof(Lista));
            if (*tail == NULL) {
                // Gestione errore di allocazione
                return;
            }
            (*tail)->value = current->value;
            (*tail)->nextPtr = NULL;
            tail = &((*tail)->nextPtr);
        }
        current = current->nextPtr;
        count++;
    }
}
```

5. Bilanciamento di un albero binario

```c
int max(int a, int b) {
    return (a > b) ? a : b;
}

int height(BTree *node) {
    if (node == NULL) return 0;
    return 1 + max(height(node->leftPtr), height(node->rightPtr));
}

int is_balanced(BTree *root) {
    if (root == NULL) return 1;

    int left_height = height(root->leftPtr);
    int right_height = height(root->rightPtr);

    if (abs(left_height - right_height) <= 1 &&
        is_balanced(root->leftPtr) &&
        is_balanced(root->rightPtr))
        return 1;

    return 0;
}
```

6. Rotazione ciclica di un array (7 punti, integrazione ≥ 6 crediti)

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

void rotate_array(int A[], int n, int k) {
    k = k % n;  // Gestisce il caso in cui k > n
    if (k == 0) return;  // Nessuna rotazione necessaria

    // Inverti l'intero array
    reverse(A, 0, n - 1);

    // Inverti i primi k elementi
    reverse(A, 0, k - 1);
```

```
    // Inverti gli ultimi n-k elementi
    reverse(A, k, n - 1);
}
```

7. Manipolazione di lista collegata

```
void remove_multiples_of_three(Lista **ptrPtr) {
    if (*ptrPtr == NULL) {
        return;
    }

    if ((*ptrPtr)->value % 3 == 0) {
        Lista *temp = *ptrPtr;
        *ptrPtr = (*ptrPtr)->nextPtr;
        free(temp);
        remove_multiples_of_three(ptrPtr);
    } else {
        remove_multiples_of_three(&((*ptrPtr)->nextPtr));
    }
}
```

8. Operazioni su albero binario di ricerca

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

    int sum = 0;
    if ((root->leftPtr == NULL && root->rightPtr != NULL) ||
        (root->leftPtr != NULL && root->rightPtr == NULL)) {
        sum += root->valore;
    }

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

9. Manipolazione di array bidimensionale

```
void swap_diagonals(int matrix[N][N]) {
    for (int i = 0; i < N; i++) {
        int temp = matrix[i][i];
```

```
        matrix[i][i] = matrix[i][N-1-i];
        matrix[i][N-1-i] = temp;
    }
}
```

10. Ricorsione su array

```
int has_increasing_sequence_helper(int A[], int size, int count) {
    if (size < 2) {
        return count >= 3;
    }

    if (A[0] < A[1]) {
        return has_increasing_sequence_helper(A + 1, size - 1, count + 1);
    } else {
        return has_increasing_sequence_helper(A + 1, size - 1, 1) ||
               (count >= 3);
    }
}

int has_increasing_sequence(int A[], int size) {
    return has_increasing_sequence_helper(A, size, 1);
}
```