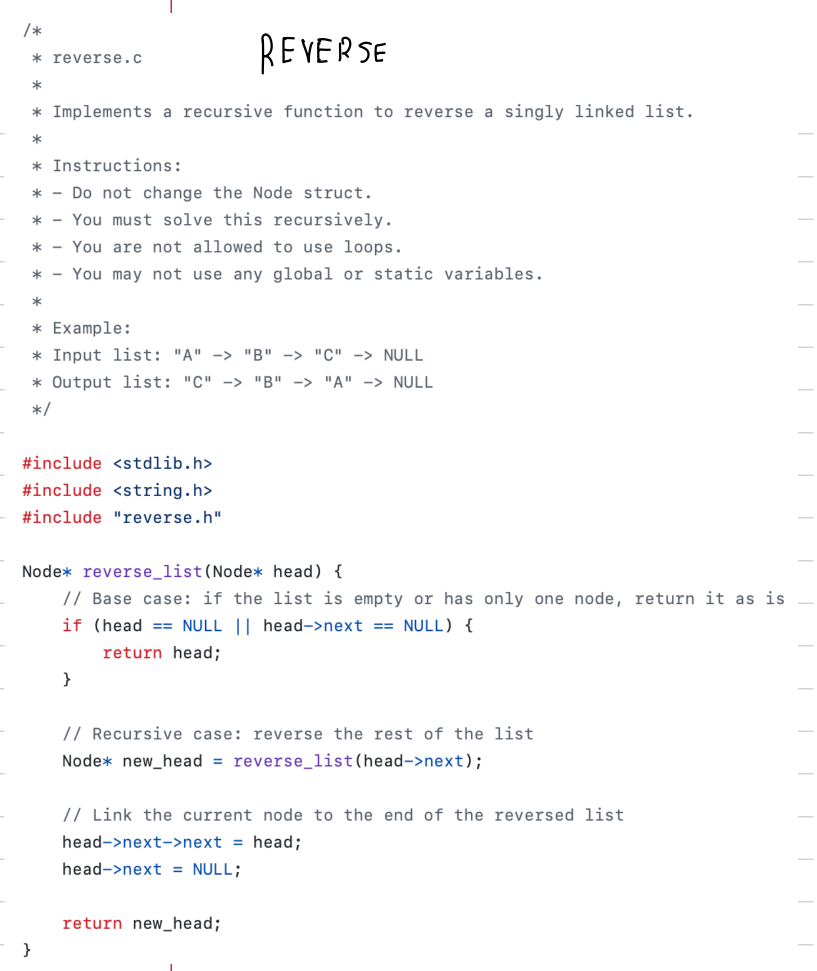
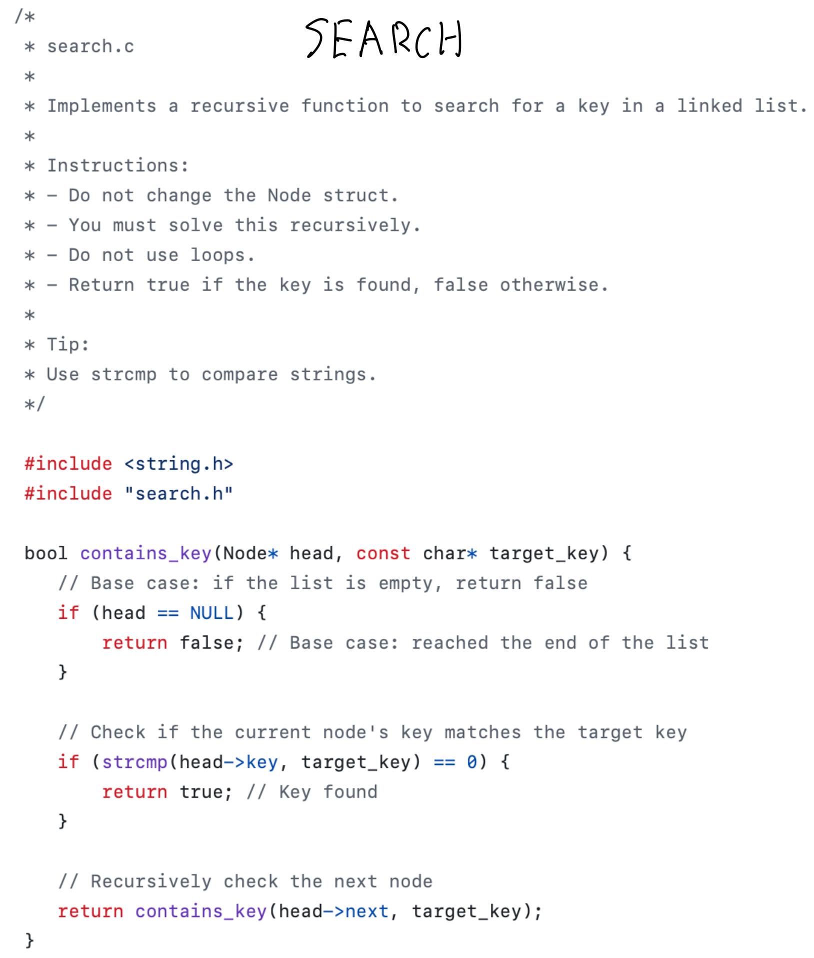
REVERSE



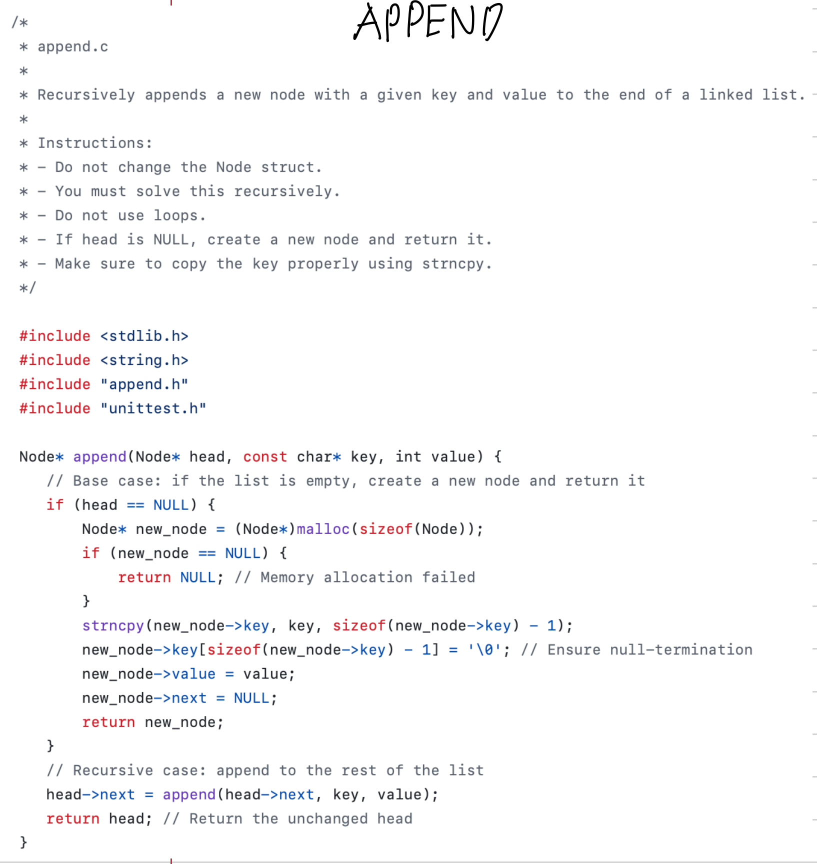
```c
/*
 * reverse.c
 *
 * Implements a recursive function to reverse a singly linked list.
 *
 * Instructions:
 * - Do not change the Node struct.
 * - You must solve this recursively.
 * - You are not allowed to use loops.
 * - You may not use any global or static variables.
 *
 * Example:
 * Input list: "A" -> "B" -> "C" -> NULL
 * Output list: "C" -> "B" -> "A" -> NULL
 */

#include <stdlib.h>
#include <string.h>
#include "reverse.h"

Node* reverse_list(Node* head) {
    // Base case: if the list is empty or has only one node, return it as is
    if (head == NULL || head->next == NULL) {
        return head;
    }

    // Recursive case: reverse the rest of the list
    Node* new_head = reverse_list(head->next);

    // Link the current node to the end of the reversed list
    head->next->next = head;
    head->next = NULL;

    return new_head;
}
```

SEARCH



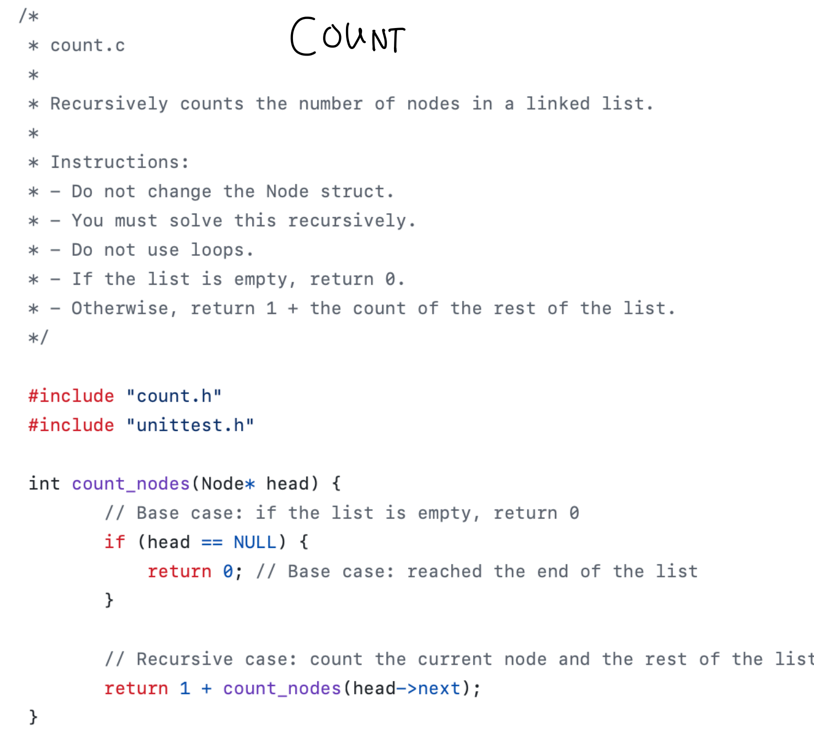
```c
/*
 * search.c
 *
 * Implements a recursive function to search for a key in a linked list.
 *
 * Instructions:
 * - Do not change the Node struct.
 * - You must solve this recursively.
 * - Do not use loops.
 * - Return true if the key is found, false otherwise.
 *
 * Tip:
 * Use strcmp to compare strings.
 */

#include <string.h>
#include "search.h"

bool contains_key(Node* head, const char* target_key) {
   // Base case: if the list is empty, return false
   if (head == NULL) {
       return false; // Base case: reached the end of the list
   }

   // Check if the current node's key matches the target key
   if (strcmp(head->key, target_key) == 0) {
       return true; // Key found
   }

   // Recursively check the next node
   return contains_key(head->next, target_key);
}
```

APPEND



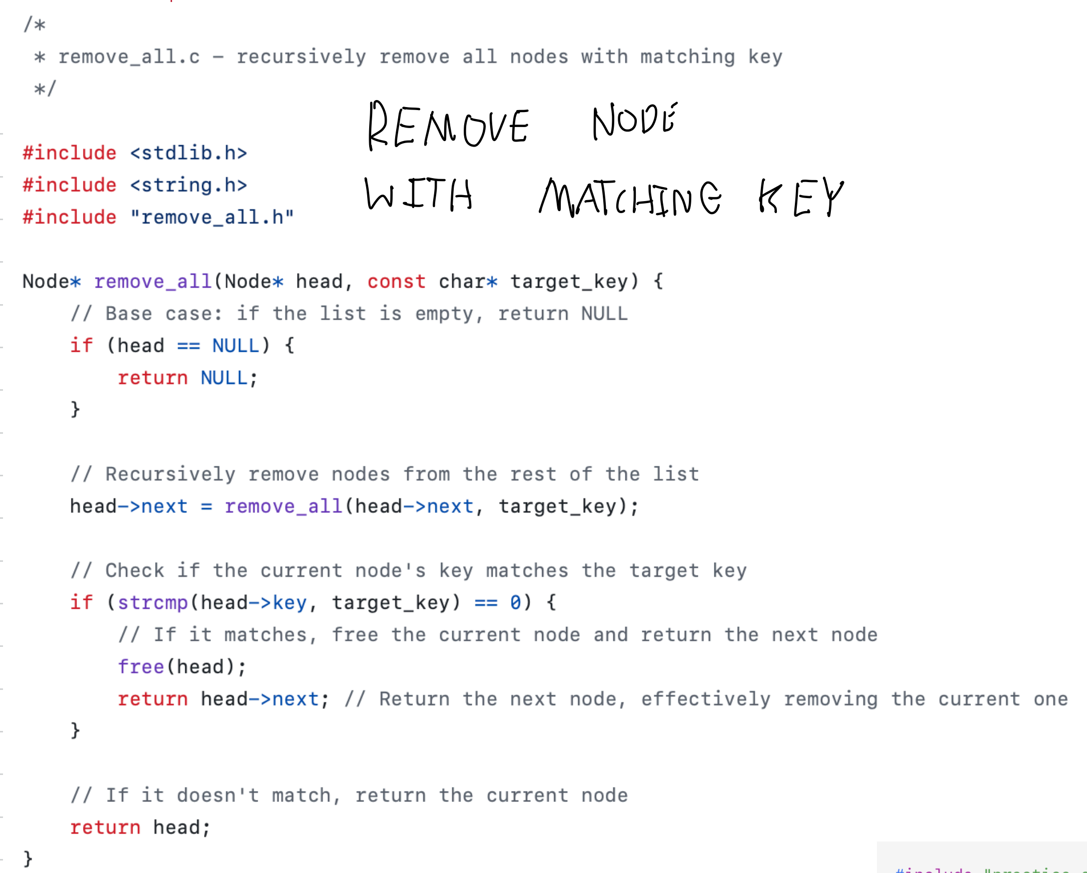
```c
/*
 * append.c
 *
 * Recursively appends a new node with a given key and value to the end of a linked list.
 *
 * Instructions:
 * - Do not change the Node struct.
 * - You must solve this recursively.
 * - Do not use loops.
 * - If head is NULL, create a new node and return it.
 * - Make sure to copy the key properly using strncpy.
 */

#include <stdlib.h>
#include <string.h>
#include "append.h"
#include "unittest.h"

Node* append(Node* head, const char* key, int value) {
   // Base case: if the list is empty, create a new node and return it
   if (head == NULL) {
       Node* new_node = (Node*)malloc(sizeof(Node));
       if (new_node == NULL) {
           return NULL; // Memory allocation failed
       }
       strncpy(new_node->key, key, sizeof(new_node->key) - 1);
       new_node->key[sizeof(new_node->key) - 1] = '\0'; // Ensure null-termination
       new_node->value = value;
       new_node->next = NULL;
       return new_node;
   }
   // Recursive case: append to the rest of the list
   head->next = append(head->next, key, value);
   return head; // Return the unchanged head
}
```

COUNT

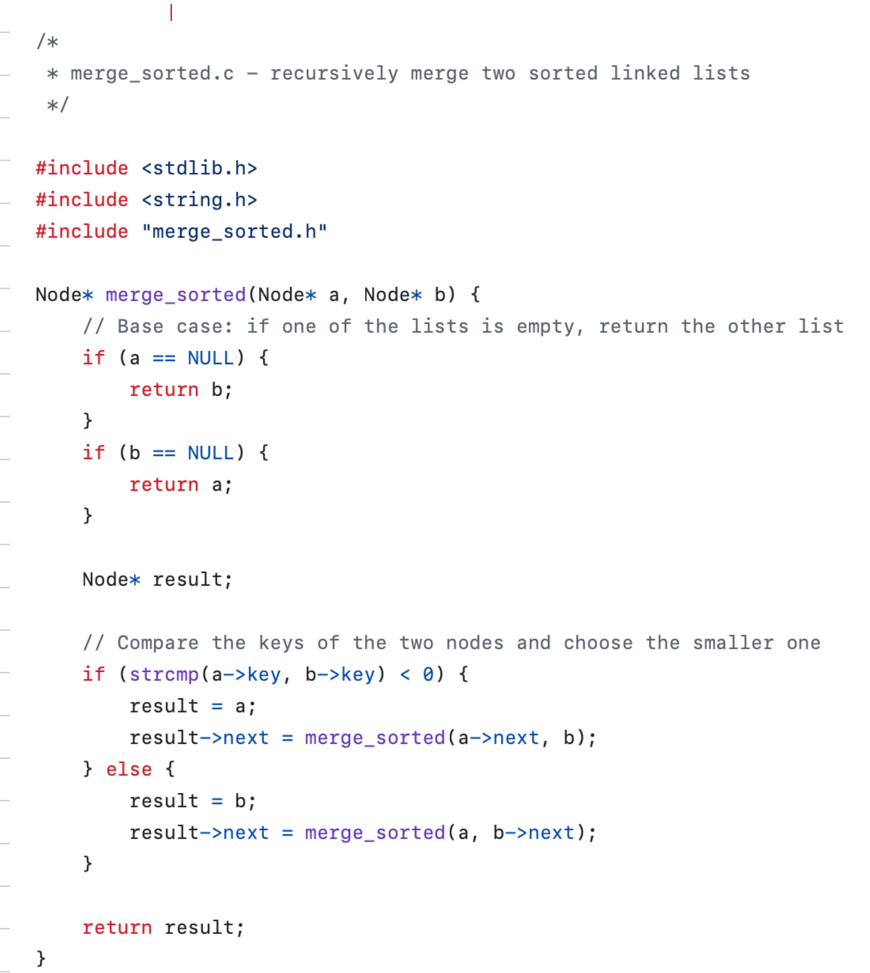


```c
/*
 * count.c
 *
 * Recursively counts the number of nodes in a linked list.
 *
 * Instructions:
 * - Do not change the Node struct.
 * - You must solve this recursively.
 * - Do not use loops.
 * - If the list is empty, return 0.
 * - Otherwise, return 1 + the count of the rest of the list.
 */

#include "count.h"
#include "unittest.h"

int count_nodes(Node* head) {
        // Base case: if the list is empty, return 0
        if (head == NULL) {
            return 0; // Base case: reached the end of the list
        }

        // Recursive case: count the current node and the rest of the list
        return 1 + count_nodes(head->next);
}
```

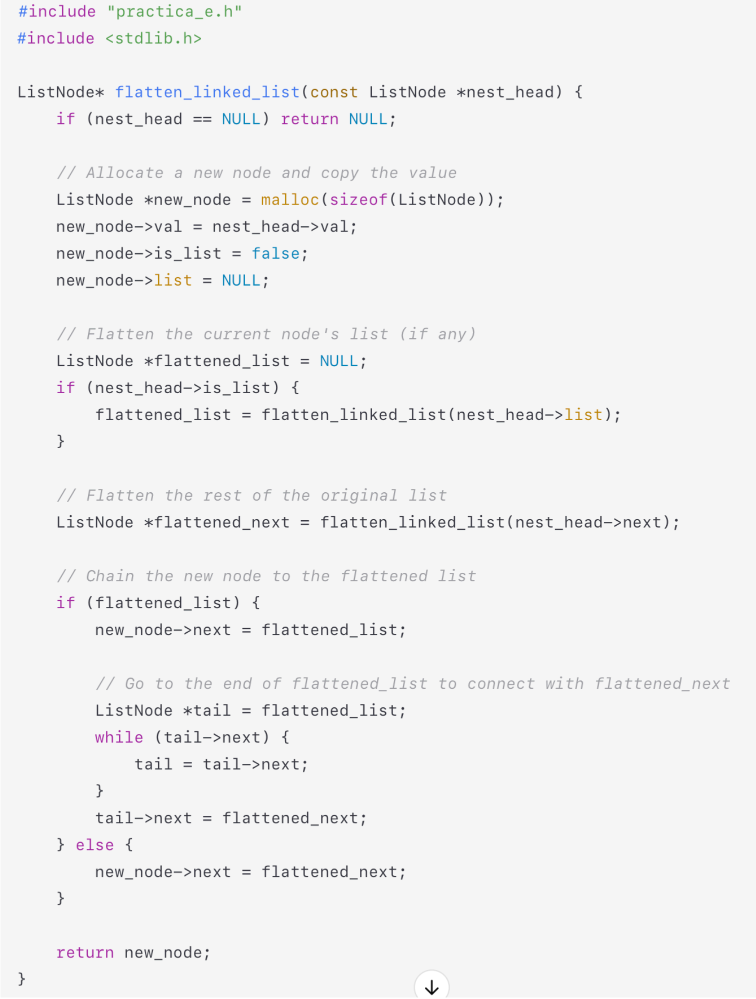
REMOVE NODE WITH MATCHING KEY



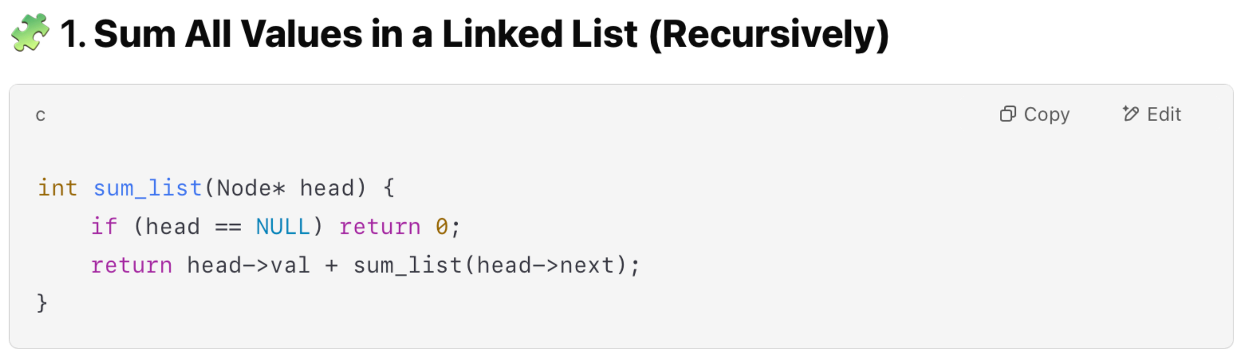
```c
/*
 * remove_all.c - recursively remove all nodes with matching key
 */

#include <stdlib.h>
#include <string.h>
#include "remove_all.h"

Node* remove_all(Node* head, const char* target_key) {
    // Base case: if the list is empty, return NULL
    if (head == NULL) {
        return NULL;
    }

    // Recursively remove nodes from the rest of the list
    head->next = remove_all(head->next, target_key);

    // Check if the current node's key matches the target key
    if (strcmp(head->key, target_key) == 0) {
        // If it matches, free the current node and return the next node
        free(head);
        return head->next; // Return the next node, effectively removing the current one
    }

    // If it doesn't match, return the current node
    return head;
}
```



```c
/*
 * merge_sorted.c - recursively merge two sorted linked lists
 */

#include <stdlib.h>
#include <string.h>
#include "merge_sorted.h"

Node* merge_sorted(Node* a, Node* b) {
    // Base case: if one of the lists is empty, return the other list
    if (a == NULL) {
        return b;
    }
    if (b == NULL) {
        return a;
    }

    Node* result;

    // Compare the keys of the two nodes and choose the smaller one
    if (strcmp(a->key, b->key) < 0) {
        result = a;
        result->next = merge_sorted(a->next, b);
    } else {
        result = b;
        result->next = merge_sorted(a, b->next);
    }

    return result;
}
```

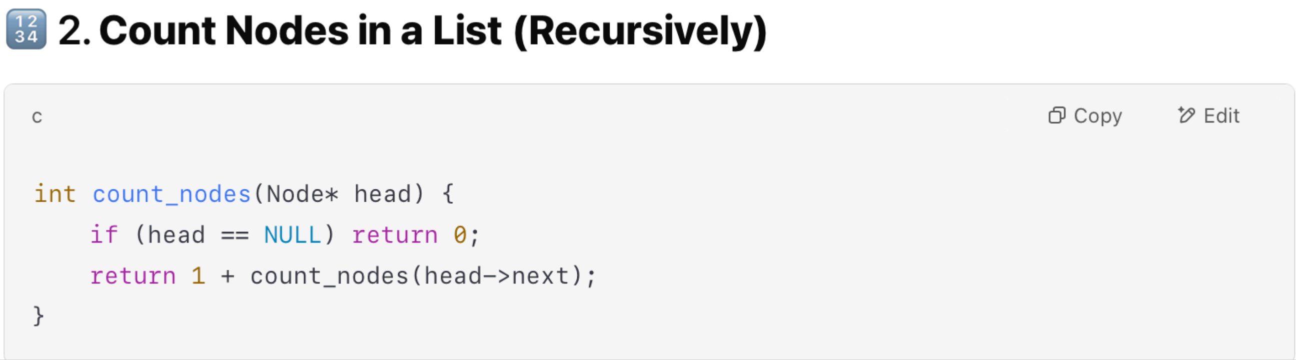


```c
#include "practica_e.h"
#include <stdlib.h>

ListNode* flatten_linked_list(const ListNode *nest_head) {
    if (nest_head == NULL) return NULL;

    // Allocate a new node and copy the value
    ListNode *new_node = malloc(sizeof(ListNode));
    new_node->val = nest_head->val;
    new_node->is_list = false;
    new_node->list = NULL;

    // Flatten the current node's list (if any)
    ListNode *flattened_list = NULL;
    if (nest_head->is_list) {
        flattened_list = flatten_linked_list(nest_head->list);
    }

    // Flatten the rest of the original list
    ListNode *flattened_next = flatten_linked_list(nest_head->next);

    // Chain the new node to the flattened list
    if (flattened_list) {
        new_node->next = flattened_list;

        // Go to the end of flattened_list to connect with flattened_next
        ListNode *tail = flattened_list;
        while (tail->next) {
            tail = tail->next;
        }
        tail->next = flattened_next;
    } else {
        new_node->next = flattened_next;
    }

    return new_node;
}
```



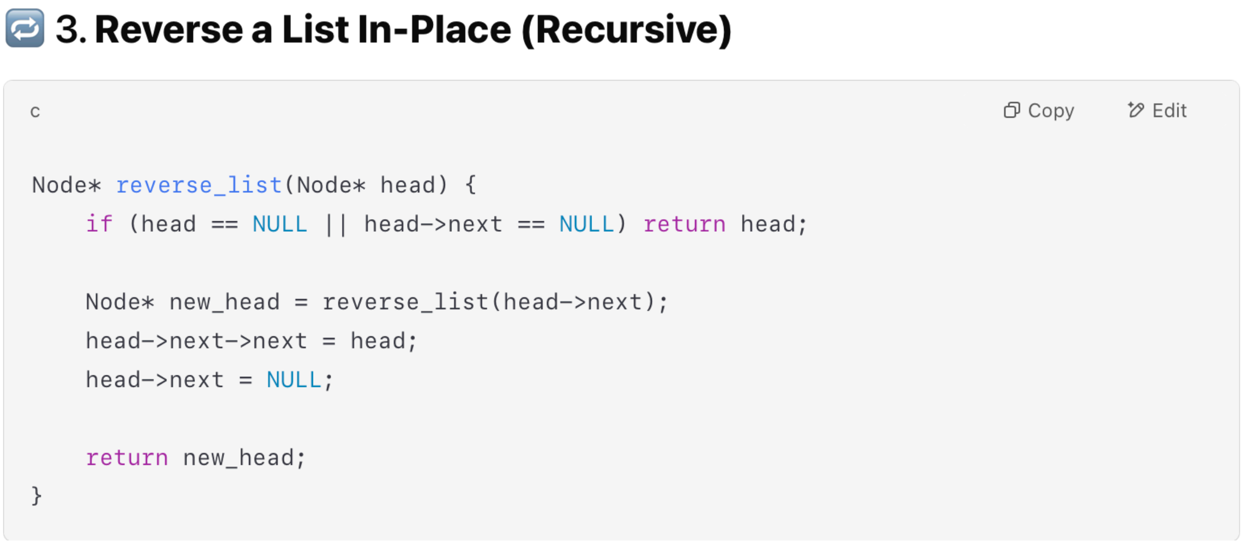
🧩 1. Sum All Values in a Linked List (Recursively)

```c
int sum_list(Node* head) {
    if (head == NULL) return 0;
    return head->val + sum_list(head->next);
}
```



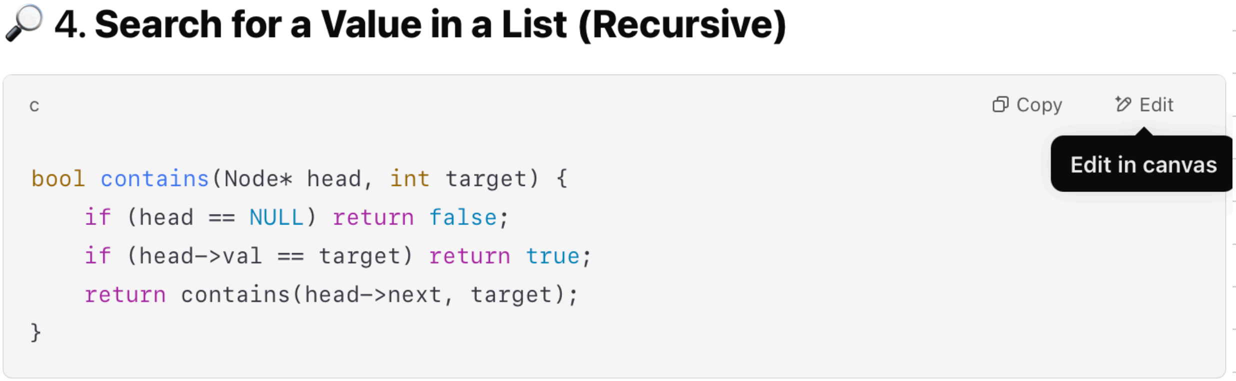
🔢 2. Count Nodes in a List (Recursively)

```c
int count_nodes(Node* head) {
    if (head == NULL) return 0;
    return 1 + count_nodes(head->next);
}
```



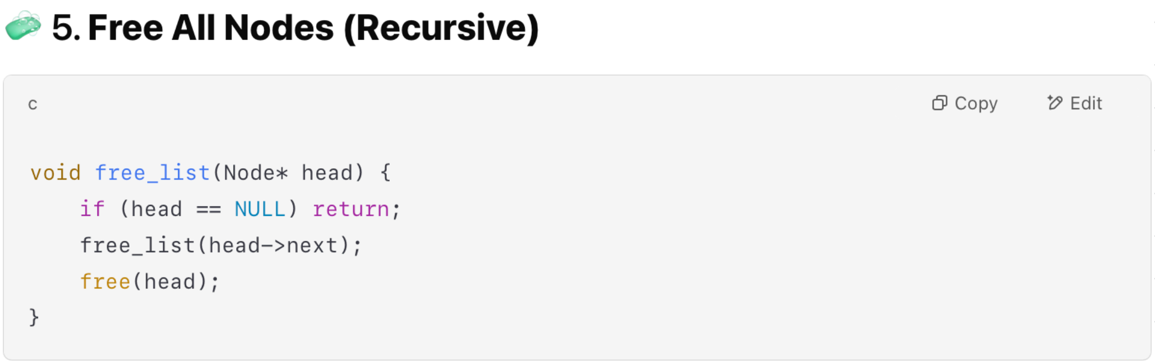
🔁 3. Reverse a List In-Place (Recursive)

```c
Node* reverse_list(Node* head) {
    if (head == NULL || head->next == NULL) return head;

    Node* new_head = reverse_list(head->next);
    head->next->next = head;
    head->next = NULL;

    return new_head;
}
```



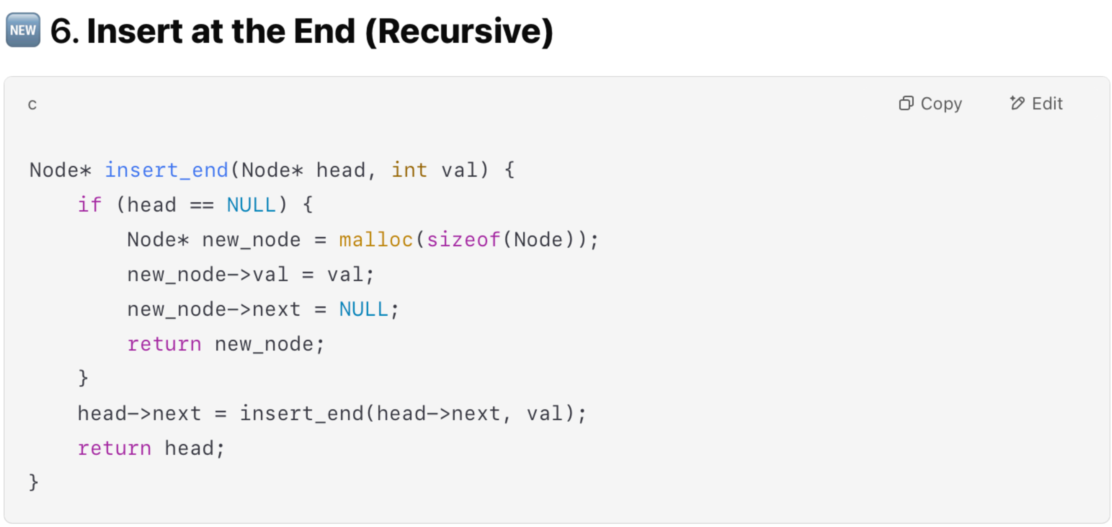
🔎 4. Search for a Value in a List (Recursive)

```c
bool contains(Node* head, int target) {
    if (head == NULL) return false;
    if (head->val == target) return true;
    return contains(head->next, target);
}
```



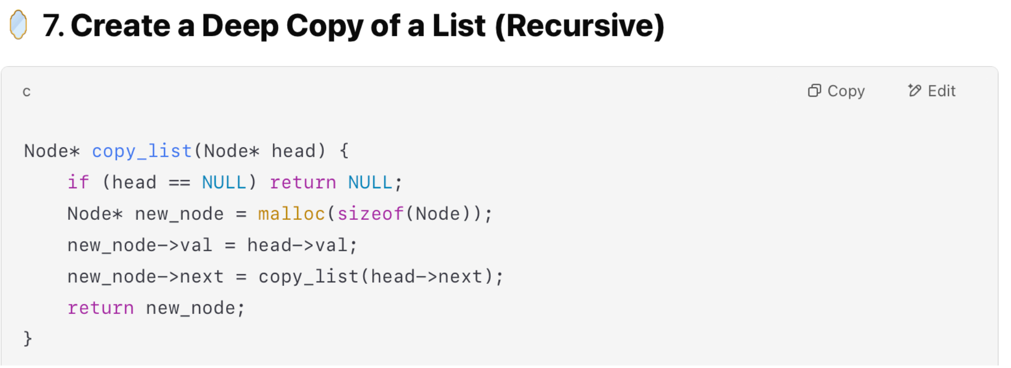
🧼 5. Free All Nodes (Recursive)

```c
void free_list(Node* head) {
    if (head == NULL) return;
    free_list(head->next);
    free(head);
}
```



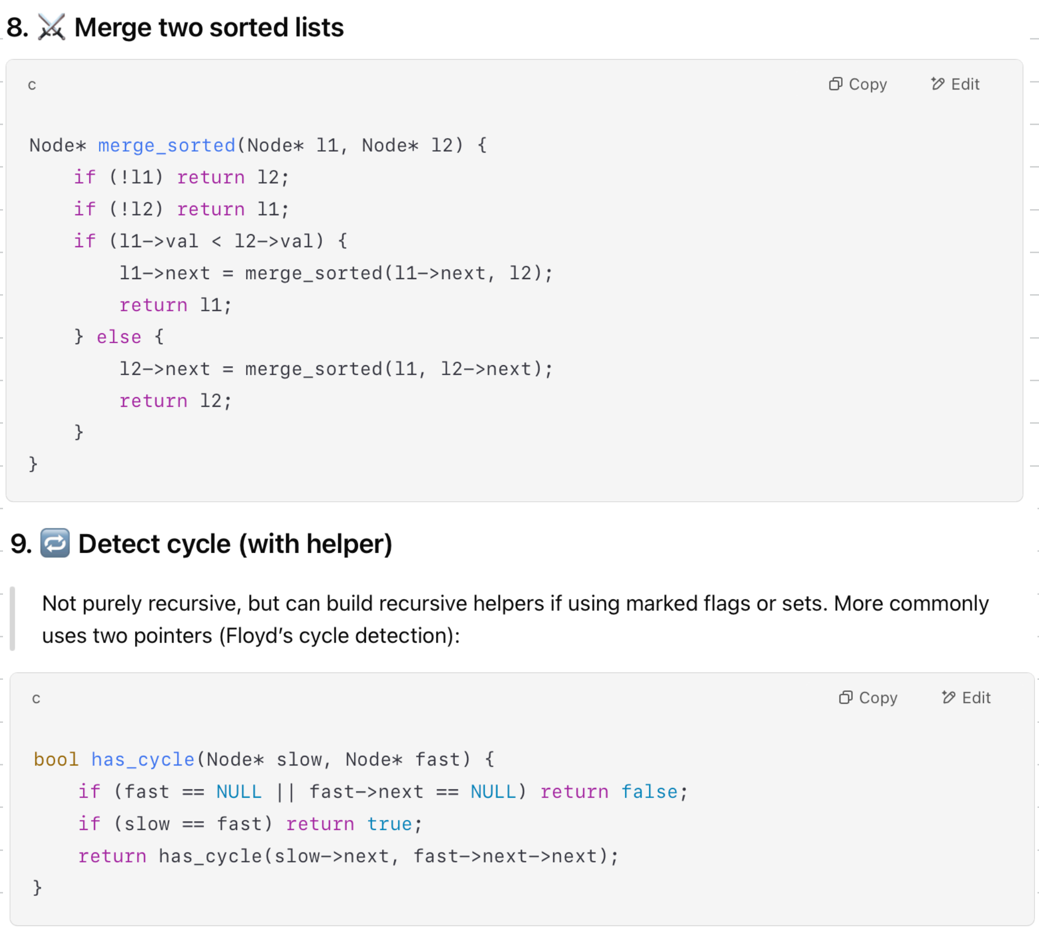
🆕 6. Insert at the End (Recursive)

```c
Node* insert_end(Node* head, int val) {
    if (head == NULL) {
        Node* new_node = malloc(sizeof(Node));
        new_node->val = val;
        new_node->next = NULL;
        return new_node;
    }
    head->next = insert_end(head->next, val);
    return head;
}
```



🪞 7. Create a Deep Copy of a List (Recursive)

```c
Node* copy_list(Node* head) {
    if (head == NULL) return NULL;
    Node* new_node = malloc(sizeof(Node));
    new_node->val = head->val;
    new_node->next = copy_list(head->next);
    return new_node;
}
```



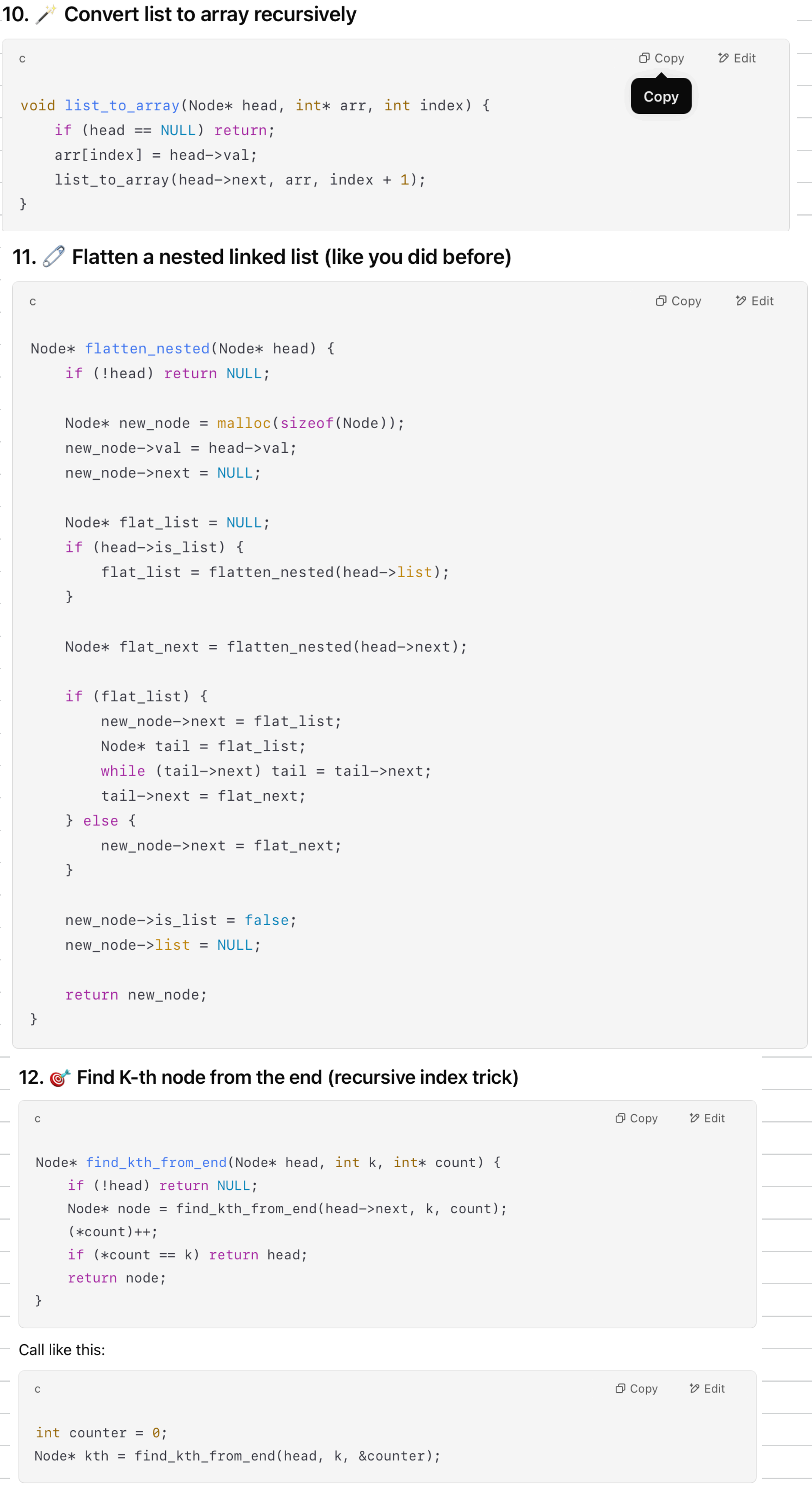
8. ⚔️ Merge two sorted lists

```c
Node* merge_sorted(Node* l1, Node* l2) {
    if (!l1) return l2;
    if (!l2) return l1;
    if (l1->val < l2->val) {
        l1->next = merge_sorted(l1->next, l2);
        return l1;
    } else {
        l2->next = merge_sorted(l1, l2->next);
        return l2;
    }
}
```

9. 🔁 Detect cycle (with helper)

> Not purely recursive, but can build recursive helpers if using marked flags or sets. More commonly uses two pointers (Floyd's cycle detection):

```c
bool has_cycle(Node* slow, Node* fast) {
    if (fast == NULL || fast->next == NULL) return false;
    if (slow == fast) return true;
    return has_cycle(slow->next, fast->next->next);
}
```



10. 🪄 Convert list to array recursively

```c
void list_to_array(Node* head, int* arr, int index) {
    if (head == NULL) return;
    arr[index] = head->val;
    list_to_array(head->next, arr, index + 1);
}
```

11. 🧷 Flatten a nested linked list (like you did before)

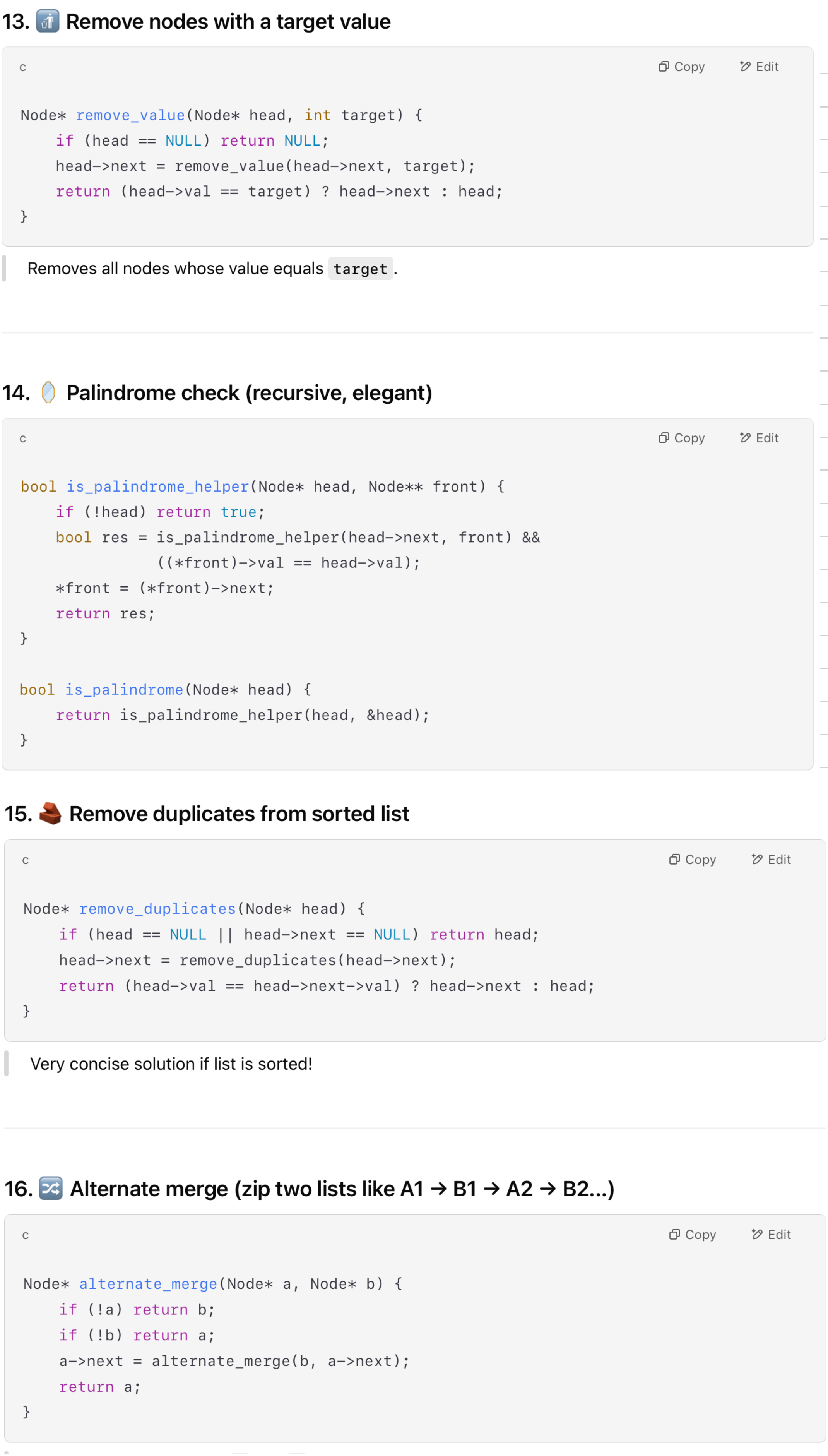
```c
Node* flatten_nested(Node* head) {
    if (!head) return NULL;

    Node* new_node = malloc(sizeof(Node));
    new_node->val = head->val;
    new_node->next = NULL;

    Node* flat_list = NULL;
    if (head->is_list) {
        flat_list = flatten_nested(head->list);
    }

    Node* flat_next = flatten_nested(head->next);

    if (flat_list) {
        new_node->next = flat_list;
        Node* tail = flat_list;
        while (tail->next) tail = tail->next;
        tail->next = flat_next;
    } else {
        new_node->next = flat_next;
    }

    new_node->is_list = false;
    new_node->list = NULL;

    return new_node;
}
```

12. 🎯 Find K-th node from the end (recursive index trick)

```c
Node* find_kth_from_end(Node* head, int k, int* count) {
    if (!head) return NULL;
    Node* node = find_kth_from_end(head->next, k, count);
    (*count)++;
    if (*count == k) return head;
    return node;
}
```

Call like this:

```c
int counter = 0;
Node* kth = find_kth_from_end(head, k, &counter);
```



13. 🚮 Remove nodes with a target value

```c
Node* remove_value(Node* head, int target) {
    if (head == NULL) return NULL;
    head->next = remove_value(head->next, target);
    return (head->val == target) ? head->next : head;
}
```

> Removes all nodes whose value equals `target`.

---

14. 🪞 Palindrome check (recursive, elegant)

```c
bool is_palindrome_helper(Node* head, Node** front) {
    if (!head) return true;
    bool res = is_palindrome_helper(head->next, front) &&
               ((*front)->val == head->val);
    *front = (*front)->next;
    return res;
}

bool is_palindrome(Node* head) {
    return is_palindrome_helper(head, &head);
}
```

15. 🧱 Remove duplicates from sorted list

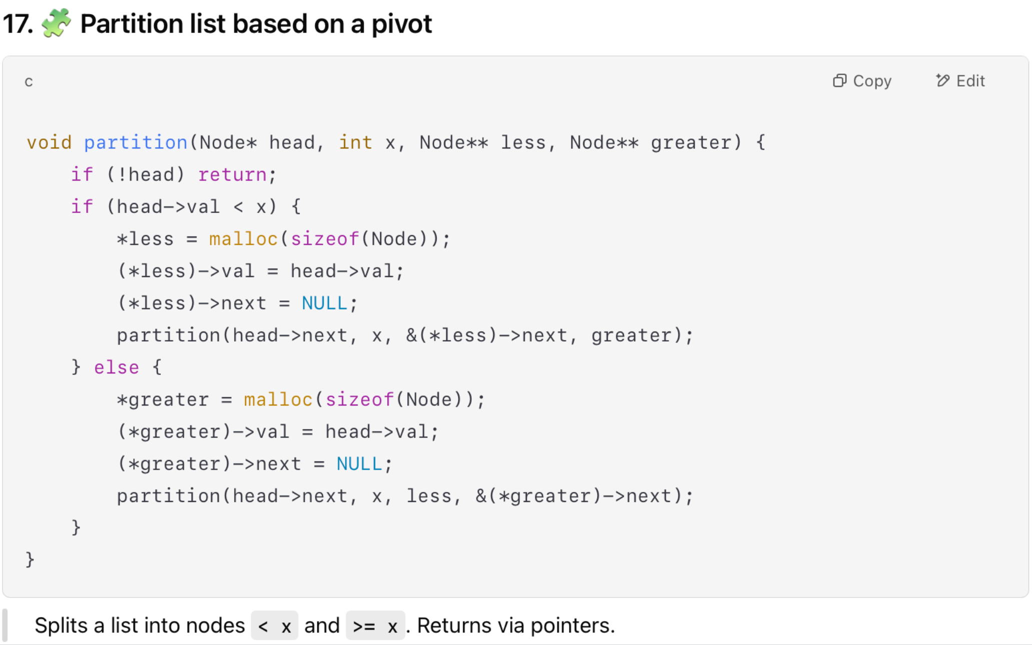
```c
Node* remove_duplicates(Node* head) {
    if (head == NULL || head->next == NULL) return head;
    head->next = remove_duplicates(head->next);
    return (head->val == head->next->val) ? head->next : head;
}
```

> Very concise solution if list is sorted!

---

16. 🔀 Alternate merge (zip two lists like A1 → B1 → A2 → B2...)

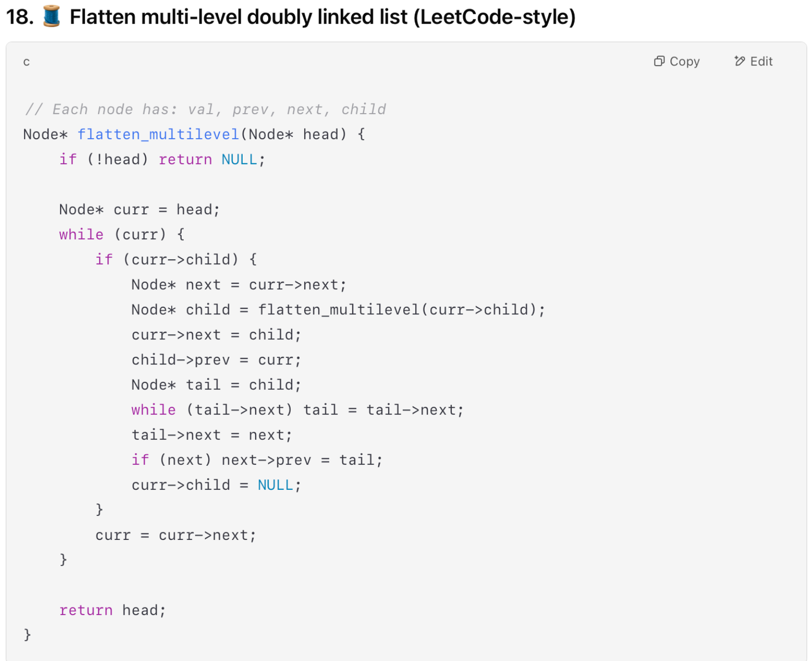
```c
Node* alternate_merge(Node* a, Node* b) {
    if (!a) return b;
    if (!b) return a;
    a->next = alternate_merge(b, a->next);
    return a;
}
```



17. 🧩 Partition list based on a pivot

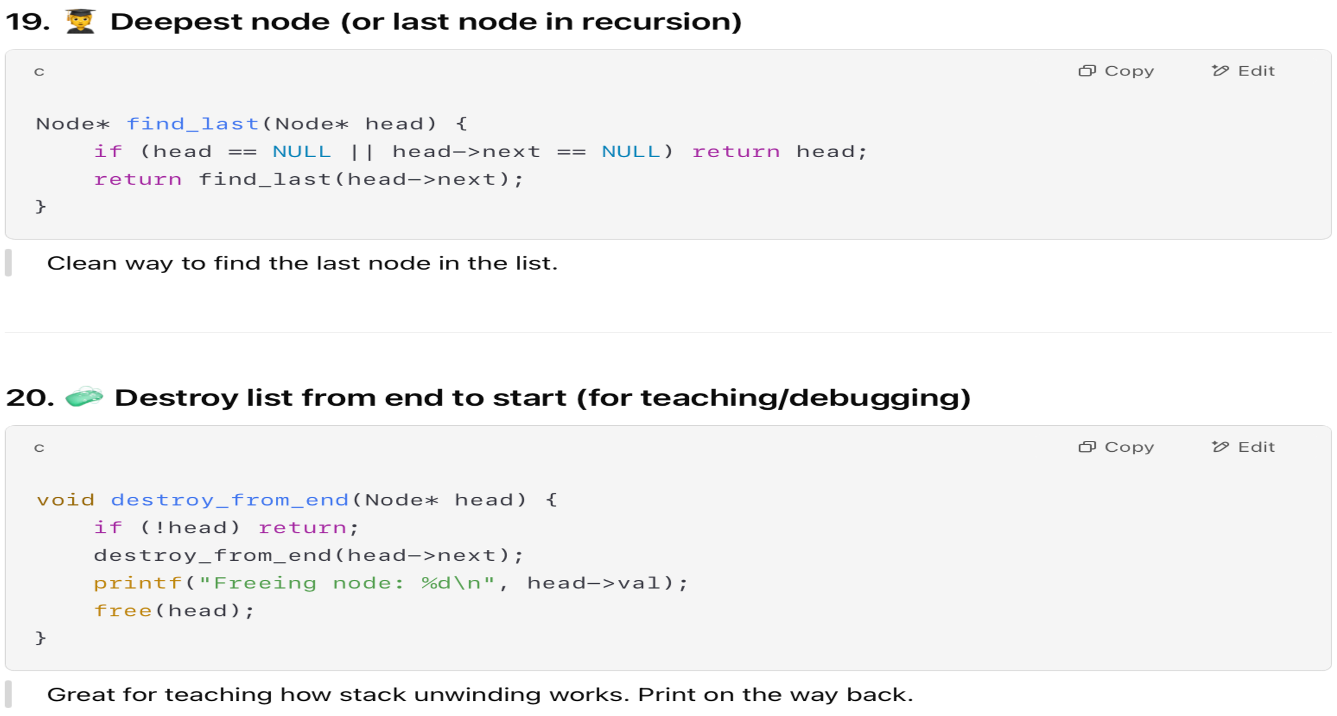
```c
void partition(Node* head, int x, Node** less, Node** greater) {
    if (!head) return;
    if (head->val < x) {
        *less = malloc(sizeof(Node));
        (*less)->val = head->val;
        (*less)->next = NULL;
        partition(head->next, x, &(*less)->next, greater);
    } else {
        *greater = malloc(sizeof(Node));
        (*greater)->val = head->val;
        (*greater)->next = NULL;
        partition(head->next, x, less, &(*greater)->next);
    }
}
```

> Splits a list into nodes `< x` and `>= x`. Returns via pointers.



18. 🧵 Flatten multi-level doubly linked list (LeetCode-style)

```c
// Each node has: val, prev, next, child
Node* flatten_multilevel(Node* head) {
    if (!head) return NULL;

    Node* curr = head;
    while (curr) {
        if (curr->child) {
            Node* next = curr->next;
            Node* child = flatten_multilevel(curr->child);
            curr->next = child;
            child->prev = curr;
            Node* tail = child;
            while (tail->next) tail = tail->next;
            tail->next = next;
            if (next) next->prev = tail;
            curr->child = NULL;
        }
        curr = curr->next;
    }

    return head;
}
```



19. 🧑‍🎓 Deepest node (or last node in recursion)

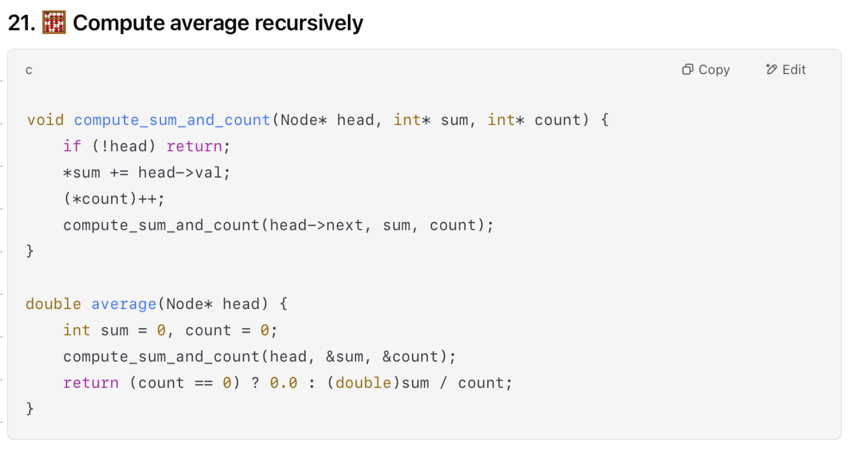
```c
Node* find_last(Node* head) {
    if (head == NULL || head->next == NULL) return head;
    return find_last(head->next);
}
```

> Clean way to find the last node in the list.

20. 🧼 Destroy list from end to start (for teaching/debugging)

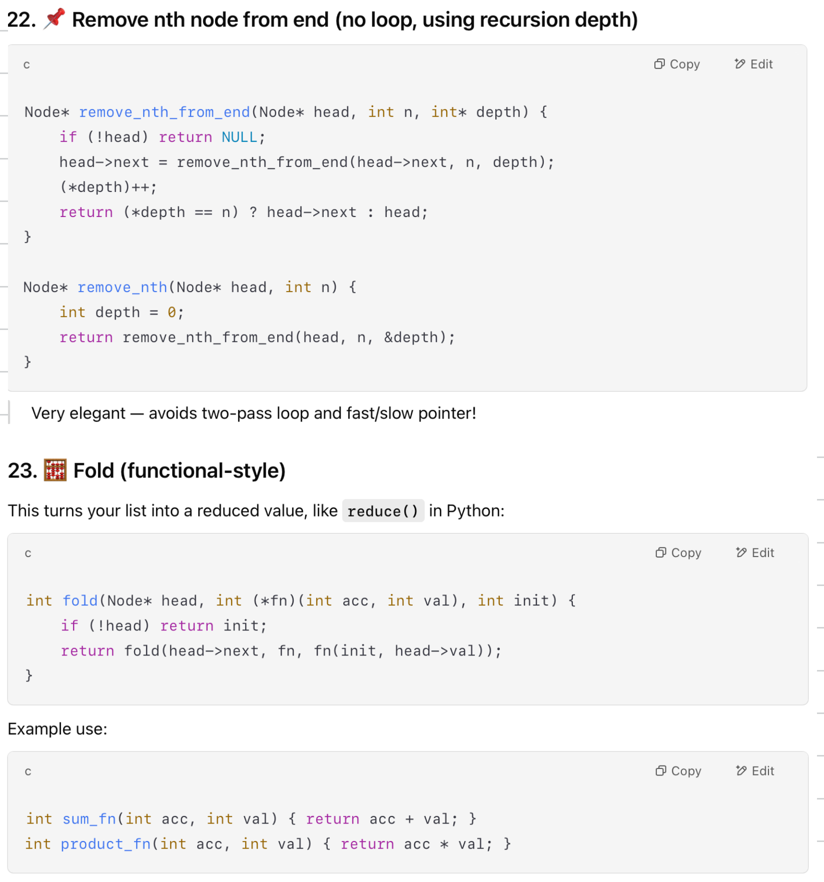
```c
void destroy_from_end(Node* head) {
    if (!head) return;
    destroy_from_end(head->next);
    printf("Freeing node: %d\n", head->val);
    free(head);
}
```

> Great for teaching how stack unwinding works. Print on the way back.



21. 🧮 Compute average recursively

```c
void compute_sum_and_count(Node* head, int* sum, int* count) {
    if (!head) return;
    *sum += head->val;
    (*count)++;
    compute_sum_and_count(head->next, sum, count);
}

double average(Node* head) {
    int sum = 0, count = 0;
    compute_sum_and_count(head, &sum, &count);
    return (count == 0) ? 0.0 : (double)sum / count;
}
```



22. 📌 Remove nth node from end (no loop, using recursion depth)

```c
Node* remove_nth_from_end(Node* head, int n, int* depth) {
    if (!head) return NULL;
    head->next = remove_nth_from_end(head->next, n, depth);
    (*depth)++;
    return (*depth == n) ? head->next : head;
}

Node* remove_nth(Node* head, int n) {
    int depth = 0;
    return remove_nth_from_end(head, n, &depth);
}
```

> Very elegant — avoids two-pass loop and fast/slow pointer!

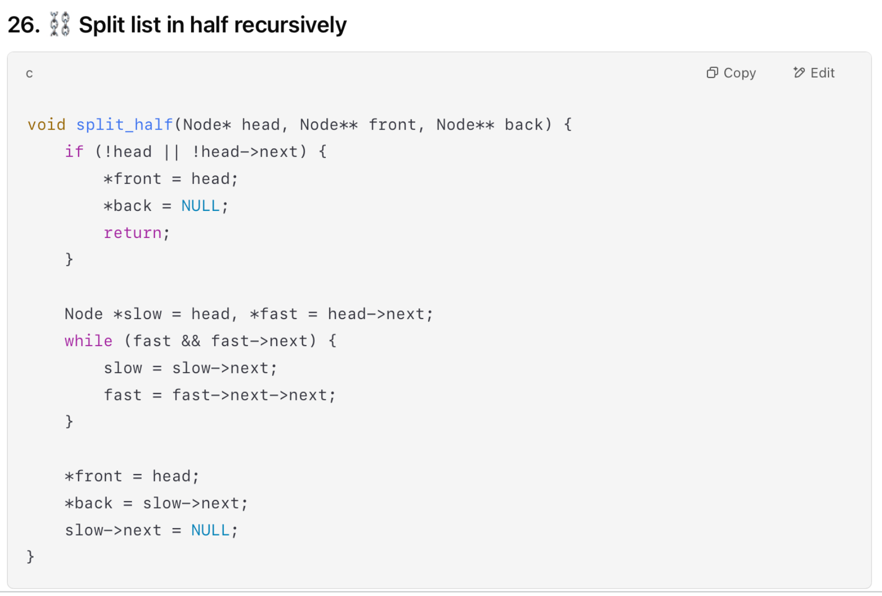
23. 🧮 Fold (functional-style)

This turns your list into a reduced value, like `reduce()` in Python:

```c
int fold(Node* head, int (*fn)(int acc, int val), int init) {
    if (!head) return init;
    return fold(head->next, fn, fn(init, head->val));
}
```

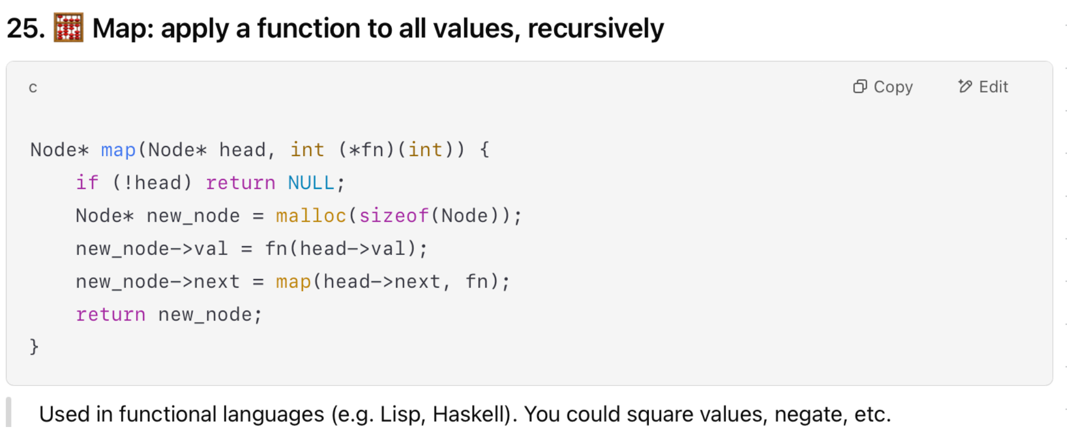
Example use:

```c
int sum_fn(int acc, int val) { return acc + val; }
int product_fn(int acc, int val) { return acc * val; }
```



26. ⛓️ Split list in half recursively

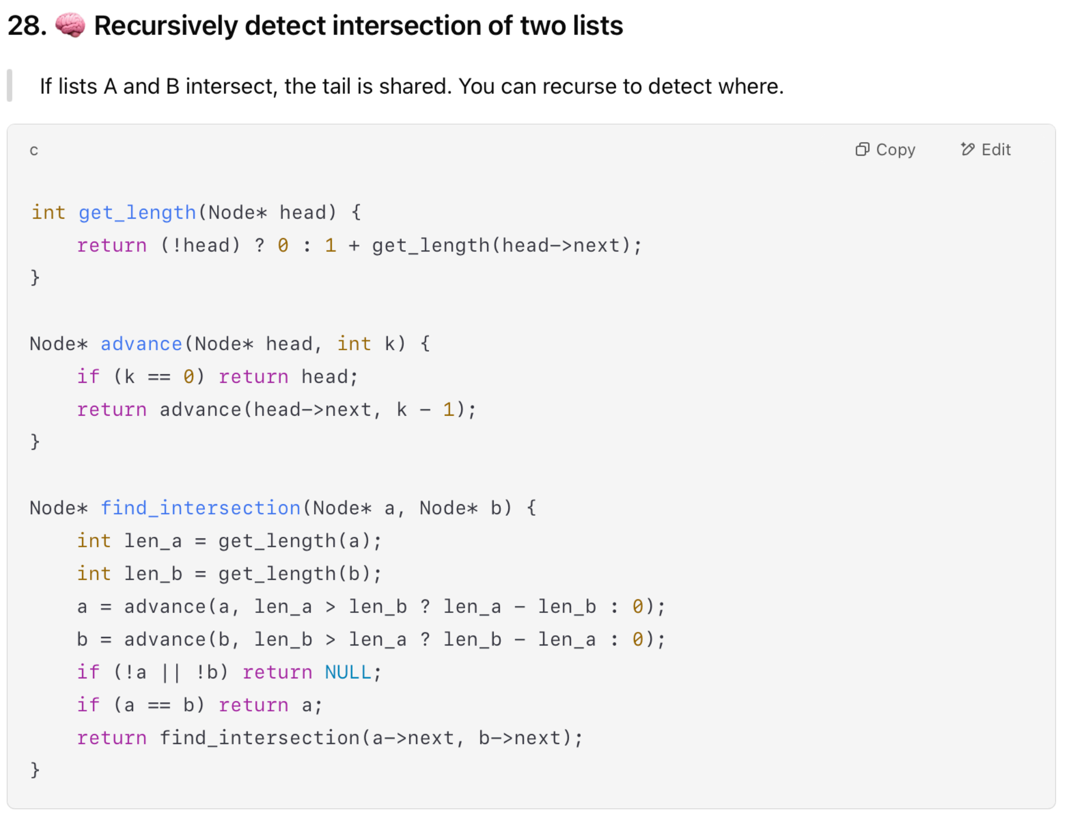
```c
void split_half(Node* head, Node** front, Node** back) {
    if (!head || !head->next) {
        *front = head;
        *back = NULL;
        return;
    }

    Node *slow = head, *fast = head->next;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }

    *front = head;
    *back = slow->next;
    slow->next = NULL;
}
```



25. 🧮 Map: apply a function to all values, recursively

```c
Node* map(Node* head, int (*fn)(int)) {
    if (!head) return NULL;
    Node* new_node = malloc(sizeof(Node));
    new_node->val = fn(head->val);
    new_node->next = map(head->next, fn);
    return new_node;
}
```

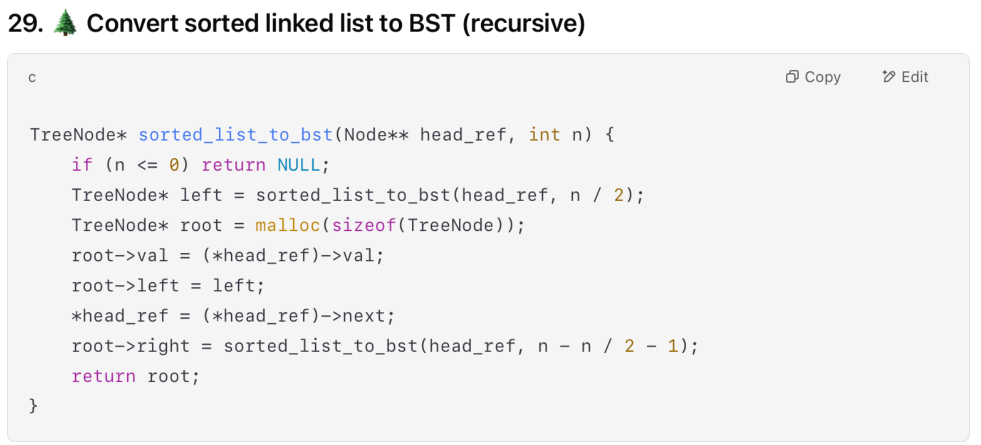
> Used in functional languages (e.g. Lisp, Haskell). You could square values, negate, etc.



28. 🧠 Recursively detect intersection of two lists

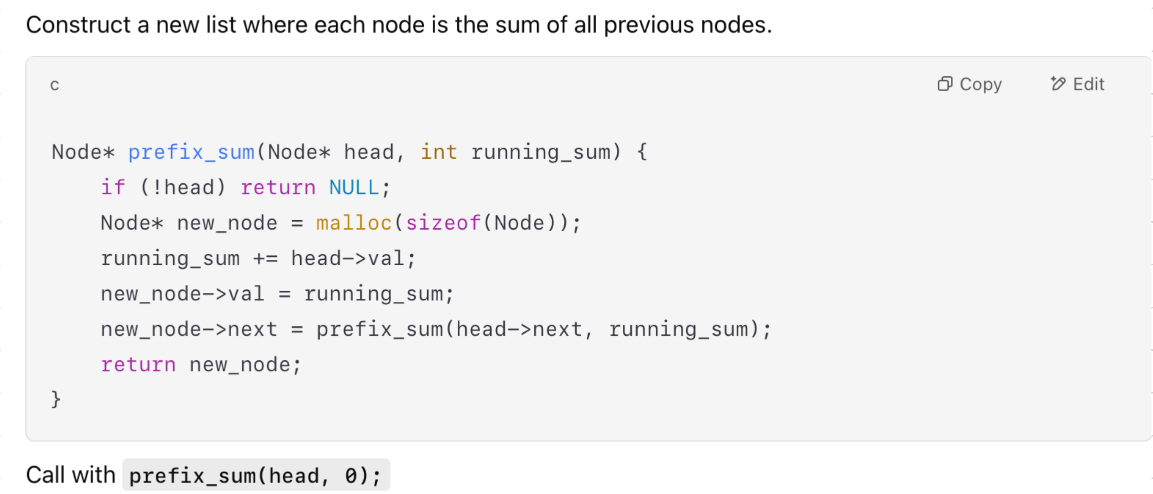
> If lists A and B intersect, the tail is shared. You can recurse to detect where.

```c
int get_length(Node* head) {
    return (!head) ? 0 : 1 + get_length(head->next);
}

Node* advance(Node* head, int k) {
    if (k == 0) return head;
    return advance(head->next, k - 1);
}

Node* find_intersection(Node* a, Node* b) {
    int len_a = get_length(a);
    int len_b = get_length(b);
    a = advance(a, len_a > len_b ? len_a - len_b : 0);
    b = advance(b, len_b > len_a ? len_b - len_a : 0);
    if (!a || !b) return NULL;
    if (a == b) return a;
    return find_intersection(a->next, b->next);
}
```



29. 🌲 Convert sorted linked list to BST (recursive)

```c
TreeNode* sorted_list_to_bst(Node** head_ref, int n) {
    if (n <= 0) return NULL;
    TreeNode* left = sorted_list_to_bst(head_ref, n / 2);
    TreeNode* root = malloc(sizeof(TreeNode));
    root->val = (*head_ref)->val;
    root->left = left;
    *head_ref = (*head_ref)->next;
    root->right = sorted_list_to_bst(head_ref, n - n / 2 - 1);
    return root;
}
```

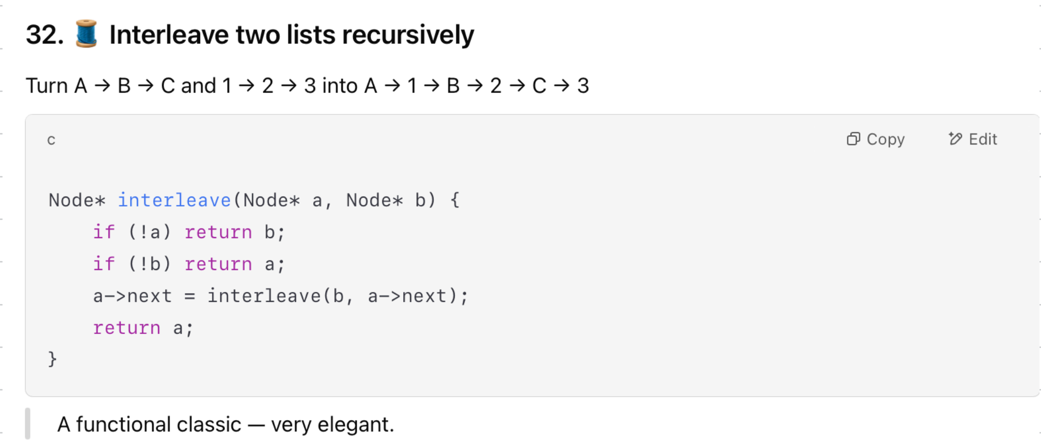


30. 🧮 Prefix sum list recursively

Construct a new list where each node is the sum of all previous nodes.

```c
Node* prefix_sum(Node* head, int running_sum) {
    if (!head) return NULL;
    Node* new_node = malloc(sizeof(Node));
    running_sum += head->val;
    new_node->val = running_sum;
    new_node->next = prefix_sum(head->next, running_sum);
    return new_node;
}
```

Call with `prefix_sum(head, 0);`

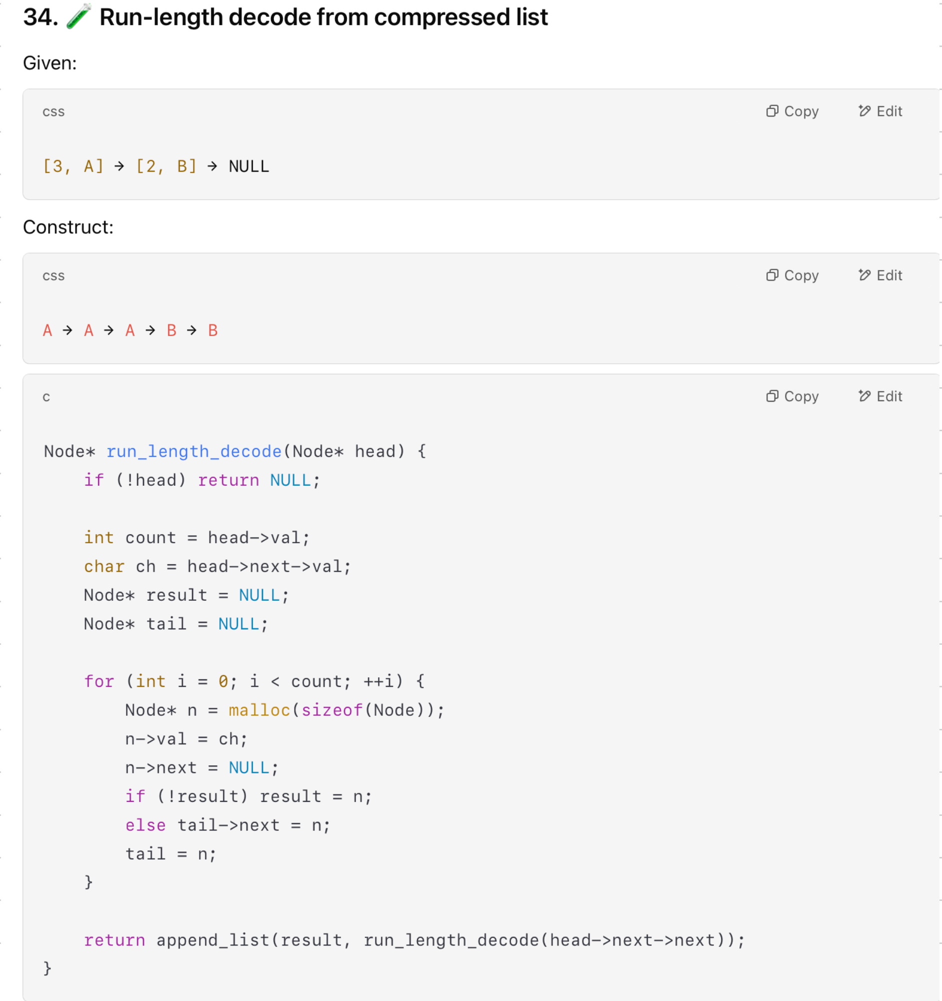


32. 🧵 Interleave two lists recursively

Turn A → B → C and 1 → 2 → 3 into A → 1 → B → 2 → C → 3

```c
Node* interleave(Node* a, Node* b) {
    if (!a) return b;
    if (!b) return a;
    a->next = interleave(b, a->next);
    return a;
}
```

> A functional classic — very elegant.



34. 🧪 Run-length decode from compressed list

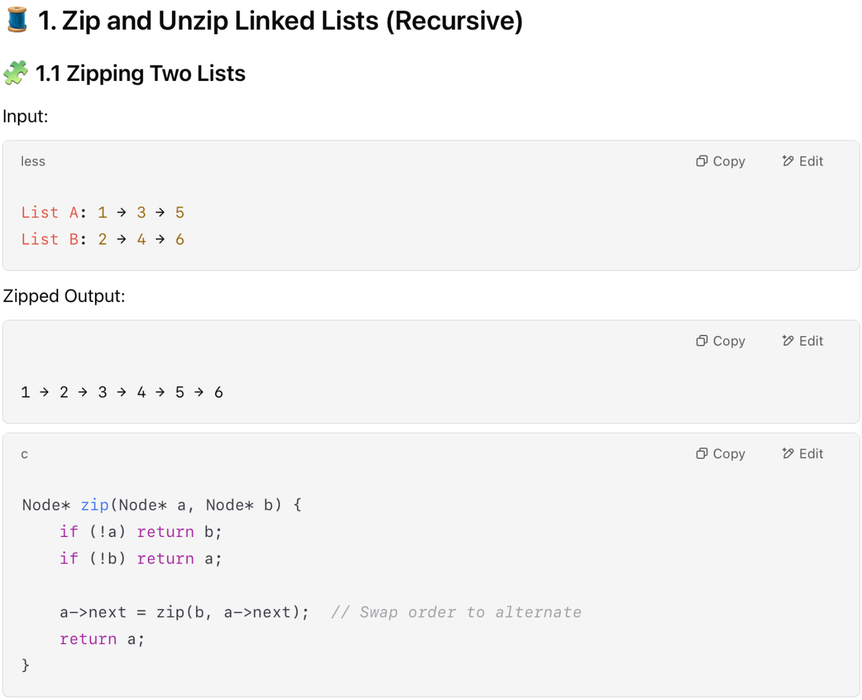
Given:

```css
[3, A] → [2, B] → NULL
```

Construct:

```css
A → A → A → B → B
```

```c
Node* run_length_decode(Node* head) {
    if (!head) return NULL;

    int count = head->val;
    char ch = head->next->val;
    Node* result = NULL;
    Node* tail = NULL;

    for (int i = 0; i < count; ++i) {
        Node* n = malloc(sizeof(Node));
        n->val = ch;
        n->next = NULL;
        if (!result) result = n;
        else tail->next = n;
        tail = n;
    }

    return append_list(result, run_length_decode(head->next->next));
}
```



🧵 1. Zip and Unzip Linked Lists (Recursive)

🧩 1.1 Zipping Two Lists

Input:

less

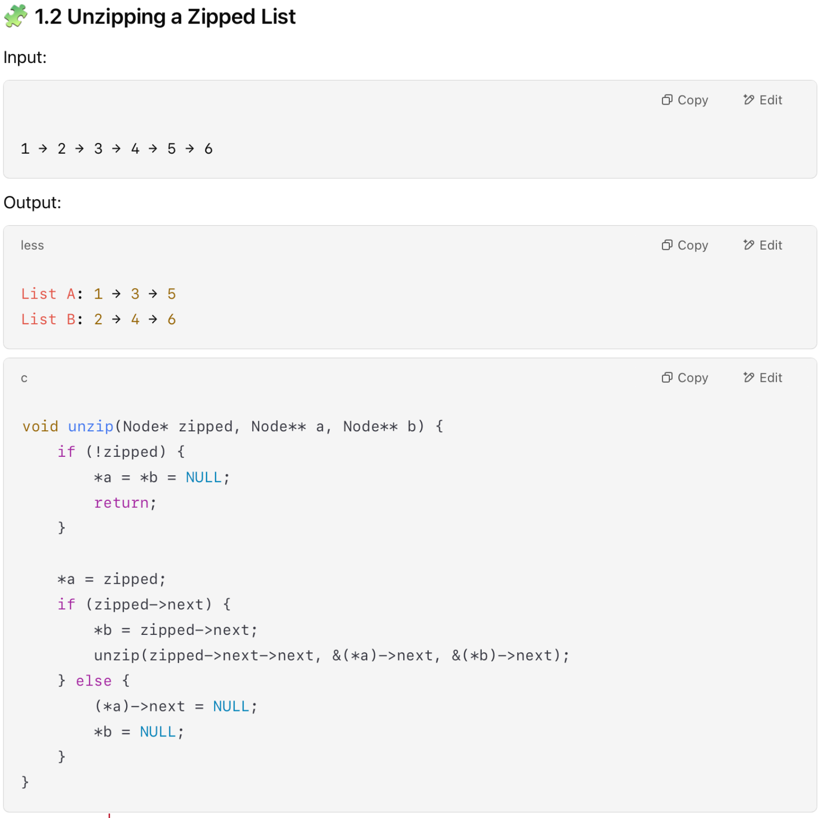
```
List A: 1 → 3 → 5
List B: 2 → 4 → 6
```

Zipped Output:

```
1 → 2 → 3 → 4 → 5 → 6
```

c

```c
Node* zip(Node* a, Node* b) {
    if (!a) return b;
    if (!b) return a;

    a->next = zip(b, a->next);  // Swap order to alternate
    return a;
}
```



🧩 1.2 Unzipping a Zipped List

Input:

```
1 → 2 → 3 → 4 → 5 → 6
```

Output:

less

```
List A: 1 → 3 → 5
List B: 2 → 4 → 6
```

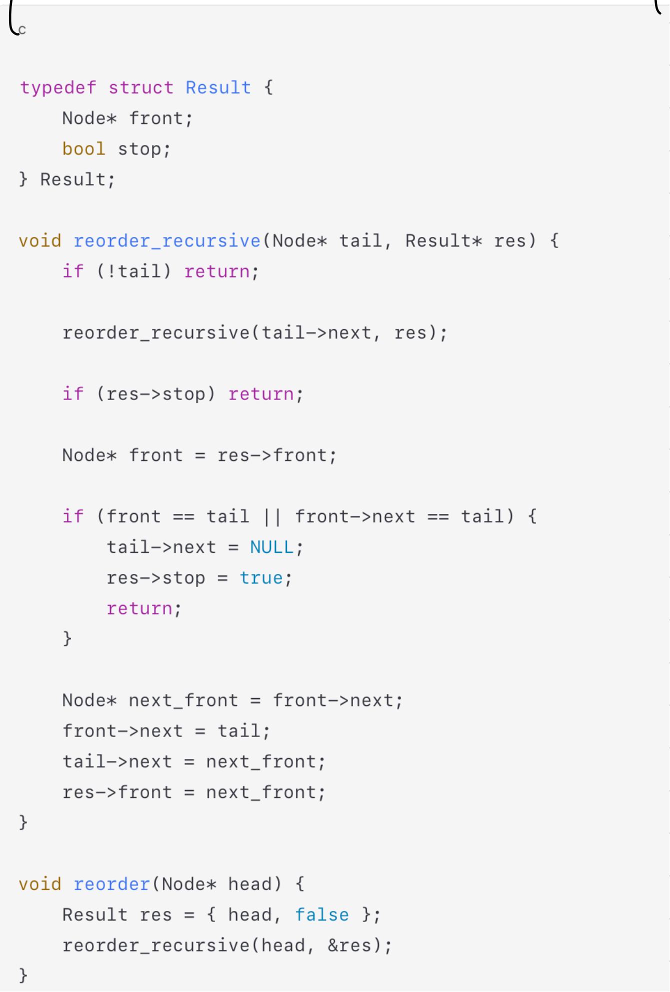
c

```c
void unzip(Node* zipped, Node** a, Node** b) {
    if (!zipped) {
        *a = *b = NULL;
        return;
    }

    *a = zipped;
    if (zipped->next) {
        *b = zipped->next;
        unzip(zipped->next->next, &(*a)->next, &(*b)->next);
    } else {
        (*a)->next = NULL;
        *b = NULL;
    }
}
```

1 → 2 → 3 → 4 → 5

1 → 5 → 2 → 4 → 3

First → last → second → second last



c

```c
typedef struct Result {
    Node* front;
    bool stop;
} Result;

void reorder_recursive(Node* tail, Result* res) {
    if (!tail) return;

    reorder_recursive(tail->next, res);

    if (res->stop) return;

    Node* front = res->front;

    if (front == tail || front->next == tail) {
        tail->next = NULL;
        res->stop = true;
        return;
    }

    Node* next_front = front->next;
    front->next = tail;
    tail->next = next_front;
    res->front = next_front;
}

void reorder(Node* head) {
    Result res = { head, false };
    reorder_recursive(head, &res);
}
```