```c
//dry part 1

#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#define SWAP_POINTERS(a, b)       \
    do {                          \
          Node tmp;               \
          tmp = a;                \
          a = b;                  \
          b = tmp;                \
    } while(0)


typedef struct node_t {
int x;
struct node_t *next;
} *Node;

typedef enum {
SUCCESS=0,
MEMORY_ERROR,
EMPTY_LIST,
UNSORTED_LIST,
NULL_ARGUMENT,
} ErrorCode;

int getListLength(Node list);
bool isListSorted(Node list);
void destroyList(Node list);
Node createNode(int x);
ErrorCode mergeSortedLists(Node list1, Node list2, Node *merged_out);


ErrorCode mergeSortedLists(Node list1, Node list2, Node *merged_out) {
// checking arguments

    if ( !getListLength(list1) || !getListLength(list2)) { //if list is
 NULL (empty) getListLength will return 0
        return EMPTY_LIST;
    }

    if ( !isListSorted(list1) || !isListSorted(list2)) {
        return UNSORTED_LIST;
    }

    if (merged_out == NULL) {
        return NULL_ARGUMENT;
    }
```

```c
//allocate new list

    *merged_out = NULL; //if the allocation failed merged_out will be N
ULL as requierd
    Node head = malloc(sizeof(*head));
    if (head == NULL) {
        return MEMORY_ERROR;
    }

//merge lists

    Node tmp = head;
    Node list1_ptr = list1;
    Node list2_ptr = list2;
    while (list1_ptr!=NULL) {
        if (list2_ptr!= NULL) {
            if (list1_ptr->x > list2_ptr->x) {
                 SWAP_POINTERS(list1_ptr, list2_ptr); //swapping betwee
n the lists if needed to copy the minimal number
            }
        }
        Node new = createNode(list1_ptr->x);
        if (new == NULL) {
            destroyList(head);
            return MEMORY_ERROR;
        }
        tmp->next = new;
        tmp = tmp->next;
        list1_ptr = list1_ptr->next;
        if (list1_ptr == NULL) {
            SWAP_POINTERS(list1_ptr, list2_ptr);
        }
    }
    *merged_out = head-
>next; //head is an empty node just for convenience so unneeded in the
final list
    free(head);
    return SUCCESS;
}

Node createNode(int x) {  //creating a new node with a given number
    Node ptr = malloc(sizeof(*ptr));
    if(!ptr) {
        return NULL;
    }
    ptr->x = x;
    ptr->next = NULL;
    return ptr;
}

void destroyList(Node ptr) {
    while(ptr) {
        Node to_delete = ptr;
        ptr = ptr->next;
        free(to_delete);
    }
}
```

```c
//dry part 2

char *stringDuplicator(char *s,int times){ // s should be str (meaningless variable name)
    assert(!s); //should be assert(s)
    assert(times > 0);
    int LEN = strlen(s); // LEN should be length due to code conventions
    char *out = malloc(LEN * times); // malloc(LEN * times + 1); need to add +1 for '\0'
    assert(out); //should check if malloc succseed and didn't return NULL, instead of assert

    //must create tmp pointer (ptr=out) so when we return @out it returns pointer to the first char and not
    // to the end of the new string
    for (int i=0; i < times; i++){
        // must switch the two lines because we want first to copy the word and then move forward
        out = out + LEN;
        strcpy(out, s); // need to check if strcpy succseed
    }
    return out;
}


char *stringDuplicator(char *str,int times){
    assert(str);
    assert(times > 0);
    int length = strlen(str);
    char *out = malloc(length * times + 1);
    if(!out) {
        return NULL;
    }
    assert(out);
    char* tmp = out;
    for (int i=0; i < times; i++){
       if(!strcpy(tmp, str)){ //strcpy failed
            free(out);
            return NULL;
       }
       tmp = tmp + length;
    }
    return out;
}
```