```c
//q2.1
//q2.1.1

#include <stdlib.h>
#include <string.h>
#include <assert.h>

char *stringduplicator(char *s, int times){  //convention1   //convention2
    assert(!s);    //coding1
    assert(times > 0);
    int LEN = strlen(*s);   //coding2   //convention3
    char *out = malloc(LEN * times);    //coding3
    assert(out);    //coding4
    for (int i = 0; i < times; i++) {
    out = out + LEN;    //convention4
    strcpy(out, s);
    }
    return out;   //coding5
}

//conventions errors-
//convention1- function name should start with upper lettter from the second word and on.
//convention2- wording function names as verbs, not action names.
//convention3- variables names should not be in capital letteres (LEN).
//convention4- no indentation is used in if block.

//coding erroes-
//coding1- if the string s isn't NULL we sholud continue the program running and not
  //collapse the program.
//coding2- strlen function input is pointer. using dereferance to a pointer sends
  //to strlen valuse instead of pointer.
//coding3- no memory allocation was made for '/0' at the end of the new srting,
  //and generally need to duplicate sizeof(len) because the allocation is of bits.
//coding4- assert check if out is NULL. if the string out is NULL we sholud return
    //NULL and not collapse the program.
//coding5- out is not a pointer for the start of the string beacuse we promoted out
    //pointer in the loop instead of promoting copy of the pointer.

//q2.1.2
//Fixed version of the above code
char *duplicateTheString(char *s, int times)
{
    if(s == NULL)
        return NULL;
    assert(times > 0);
    int len = strlen(s);
    char *out = malloc(sizeof(len) * times + 1 );
    if (out == NULL)
        return NULL;
    char *out_copy = out;
    for (int i = 0; i < times; i++)
    {
        strcpy(out_copy, s);
        out_copy = out_copy + len;
    }
    return out;
}
```

```c
//dry part 2.1.2
//sorted_linked_list.h

#ifndef SORTED_LINK_LIST
#define SORTED_LINK_LIST
#include <stdbool.h>

typedef struct node_t *Node;

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

int getListLength(Node list);

bool isListSorted(Node list);
```

```c
/*merge two sorted lists
/* Return valus-
    MEMORY_ERROR - if memory allocation failed
    EMPTY_LIST -if one of the lists is NULL
    UNSORTED_LIST - if one of the lists was unsorted
    NULL_ARGUMENT - if the merge pointer is NULL
    SUCCESS - if the merged list was created successfully
ErrorCode mergeSortedLists(Node list1, Node list2, Node *mergedOut);

/*destroy the list*/
void destroyList(Node list);

/* create new node with minimal value.
Return valus-
    NULL - if memory allocation failed
    ptr - ptr to the new node with the given value*/
Node createNode(int value);

/* Return valus- if list is NULL return new_node,
else add new_node to the end of list and return pointer to the head of list
 Error values- NULL if create next node was failed*/
Node addNodeToMergedList (Node head, Node new_node);


#endif
```

```c
//sorted_linked_list.c
#include "sorted_linked_list.h"
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

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

Node createNode(int value)
{
    Node ptr = (Node)malloc(sizeof(*ptr));
    if (ptr == NULL)
        return NULL;
    ptr->x = value;
    ptr->next= NULL;
    return ptr;
}

Node addNodeToMergedList(Node head, Node new_node)
{
    if(head == NULL)
        return new_node;
    Node ptr = head;
    while(ptr->next){
        ptr = ptr->next;
    }
    ptr->next = new_node;
    return head;
}

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

```c
ErrorCode mergeSortedLists(Node list1, Node list2, Node *mergedOut)
{
    if  (mergedOut == NULL) {
        return NULL_ARGUMENT;
    }
    if (list1 == NULL || list2 == NULL){
        mergedOut = NULL;
        return EMPTY_LIST;
    }
    Node ptr_list1= list1, ptr_list2= list2;
    if (isListSorted(ptr_list1) == false || isListSorted(ptr_list2) == false){
        return UNSORTED_LIST;
    }
    Node new_list = NULL;
    int next_smaller_value;
    while(ptr_list1 != NULL || ptr_list2 != NULL)
    {
        if((ptr_list1 == NULL && ptr_list2 != NULL) || ptr_list1->x > ptr_list2->x)
        {
            next_smaller_value = ptr_list2->x;
            ptr_list2++;
        }
        else {
            next_smaller_value = ptr_list1->x;
            ptr_list1++;
        }
        Node next_node = createNode(next_smaller_value);
        if(next_node == NULL){
            destroyList(new_list);
            mergedOut = NULL;
            return MEMORY_ERROR;
        }
        new_list = addNodeToMergedList (new_list, next_node);
    }
    mergedOut = new_list;
    return SUCCESS;
}
```