```c
#include <stdlib.h>
#include <stdbool.h>

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);
/**
* nodeListDestroy: deallocates an existing list
*
* @param ptr - target list to be deallocated from ptr to the end of the list
*/
void nodeListDestroy(Node ptr)
{
	while (ptr)
	{
		Node to_delete = ptr;
		ptr = ptr->next;
		free(to_delete);
	}
}

/**
* nodeCreateWithValue: creats new node at the end of list with a given 'x' value
* and defining perv to point at it
*
* @param last – points to the last node of the list
* @param value - 'x' value for the new node
* @return
*   MEMORY_ERROR if new node allocation failed
*   SUCCESS otherwise
*/
ErrorCode nodeCreateWithValue(Node* last, int value)
{
	assert(last);
	Node new_node = malloc(sizeof(*new_node));
	if (!new_node) {
		return MEMORY_ERROR;
	}
	new_node->next = NULL;
	new_node->x = value;
	(*last)->next = new_node;
	(*last) = (*last)->next;
	return SUCCESS;
}
```

```
/**
* nodeListCopy: copy the source list to destenation using nodeCreateWithValue
*
* @param destenation - points to the target list
* @param source - list to be copied
* @return
*   MEMORY_ERROR if new node allocation in nodeCreateWithValue failed
*   SUCCESS otherwise
*/
ErrorCode nodeListCopy(Node* destenation, Node source)
{
	for (; source != NULL; source = source->next)
	{
		if (nodeCreateWithValue(destenation, source->x) == MEMORY_ERROR)
		{
			return MEMORY_ERROR;
		}
	}
	return SUCCESS;
}

/**
* mergeSortedLists: merge two sorted lists
*
* @param list - sorted lists to merge
* @param merged_out - points to the output merged list
* @return
*   EMPTY_LIST if lists are NULL or list length is 0
*   UNSORTED_LIST if the input lists are not sorted
*   NULL_ARGUMENT if merged_out is NULL - doesn't point to target list
*   in all the cases above – content of merged_out is NULL
*   SUCCESS if merge succeeded and merged_out points to the merged list
*/
ErrorCode mergeSortedLists(Node list1, Node list2, Node* merged_out)
{
	if (merged_out == NULL) {
		return NULL_ARGUMENT;
	}

	if (list1 == NULL || !getListLength(list1)
		|| list2 == NULL || !getListLength(list2)) {
		*merged_out = NULL;
		return EMPTY_LIST;
	}

	if (!isListSorted(list1) || !isListSorted(list2)) {
		*merged_out = NULL;
		return UNSORTED_LIST;
	}

	Node head = malloc(sizeof(*head)); // helps to build and organize the merged list
	if (!head) {
		*merged_out = NULL;
		return MEMORY_ERROR;
	}
	head->next = NULL;
	*merged_out = head; // saves the beginning of merged list
```

```
        Node ptr_copy_min;
        bool first_value = true;

        while (list1 && list2) // merge lists until one of the list ended
        {
                if (list1->x < list2->x)
                {
                        ptr_copy_min = list1;
                        list1 = list1->next;
                }
                else
                {
                        ptr_copy_min = list2;
                        list2 = list2->next;
                }
                if (first_value)
                {
                        head->x = ptr_copy_min->x;
                        first_value = false;
                }
                else if (nodeCreateWithValue(&head, ptr_copy_min->x) == MEMORY_ERROR) {
                        nodeListDestroy(*merged_out);
                        *merged_out = NULL;
                        return MEMORY_ERROR;
                }

        }
        // copy the rest of the other list
        Node ptr_copy_rest;
        if (list1 == NULL)
        {
                ptr_copy_rest = list2;
        }
        else
        {
                ptr_copy_rest = list1;
        }

        ErrorCode finel_err = nodeListCopy(&head, ptr_copy_rest);
        if (finel_err == MEMORY_ERROR)
        {
                nodeListDestroy(*merged_out);
                *merged_out = NULL;
        }
        return finel_err;
}
```

תרגיל יבש שאלה 2 – שגיאות בתוכנית המקורית

שגיאות קונבנציה :

1. שם הפונקציה לא מנוסח כפועל
2. שם המשתנה LEN באותיות גדולות
3. משתנה לפונקציה s- ולא str\string או משהו יותר ברור
4. הסוגר הפותח והסוגר הסוגר של בלוק פונקציה לא מופיעים כל אחד בשורה נפרדת

שגיאות תכנותיות :

1. **הקצאה לא נכונה של out :** שורה 9 – הקצאה של out צריכה להיות בגודל LEN*times+1 לטובת 0/.
2. **שימוש לא נכון בassert :** שורה 5 – assert יוודא שתנאי מתקיים – והתנאי שניתן לו הוא "וידוא שהמחרוזת ריקה" – שזאת כמובן <u>ההפך</u> מהמטרה שלנו, שורה 10 – וידוא כישלון malloc באמצעות assert – זוהי בדיקה שצריכה להתבצע בעת ריצת קוד ולא רק בדיבוג, יש לצאת מהתוכנית אם ההקצאה לא הצליחה כדי למנוע גישה לערך שלא הוקצה.
3. **חריגה מתאים שהוקצו בזיכרון :** שורה 12 – קידום כתובת out עבור האיטרציה הראשונה מבלי שהעתקנו את המילה בפעם הראשונה ולמרות זאת הלולאה מתבצעת times פעמים – כלומר נחרוג בLEN תאים מהמילה שהקצנו.
4. **"איבוד" תאים שהוקצו :** שורה 12 – קידום כתובת out גוררת איבוד תחילת מחרוזת שהקצנו – תאים שלא נוכל למחוק. שורה 15 – כפועל יוצא של השגיאה ב3 – אנחנו לא מחזירים את תחילת out ובכך לא נחזיר את המחרוזת הכוללת.

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

/**
* duplicateString: function concatenate str to itself for "times" times
*
* @param str - string to concatenate
* @param times - defines how many concatenations to execute
* @return
*      NULL if an error occurred during the function run
*      char* with "times" "str" in a row if succeeded
*
*/


char* duplicateString(char* str, int times)
{
       assert(str);
       assert(times > 0);
       int len = strlen(str);
       char* str_out = malloc(sizeof(char) * ((times * len) + 1));
       if (!str_out) {
              return NULL;
       }
       for (int i = 0; i < times; i++) {
              strcpy(str_out + i * len, str);
       }
       return str_out;
}
```