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

char *stringduplicator(char *s, int times) {
    assert(!s);
    assert(times > 0);
    int LEN = strlen(*s);
    char *out = malloc(LEN * times);
    assert(out);
    for (int i = 0; i < times; i++) {
    out = out + LEN;
    strcpy(out, s);
    }
    return out;
}
```

2.1.1 סעיף א:

שגיאות קומפילציה

שגיאה 1 • בשם הפונקציה: האות במילה השנייה הוא לא אות גדולה

שגיאה 2 • שם הפונקציה צריך להיות פחות מאשר - duplicateString

שגיאה 3 • שם המשתנה LEN באותיות גדולות

שגיאה 4 • אין הצהרה בלולאות ה- For.

שגיאות תכנות:

שגיאה 1 • ב- assert הראשון, אם s == Null התכנית תמשיך.

שגיאה 2 • ב- strlen נשלח s* שהוא Char במקום s שהוא Char*

שגיאה 3 • ב- malloc צריך להקצות בית נוסף עבור "\0"

שגיאה 4 • צריך לוודא ש: out != Null וה- assert לא יעבוד כמו כלי דיבוג

שגיאה 5 • בלולאת For out מתקדם כל איטרציה ובסוף כשמחזירים את out

מצביעים על המקום התחלה.

2 pro 2.1.2

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

char *duplicateString(char *s, int times) {
    if(s == NULL)
        return NULL;
    assert(times > 0);
    int len = strlen(s);
    char *out = malloc(len * times + 1);
    if(out == NULL)
        return NULL;
    int j = 0;
    for(int i = 0; i < times; i++) {
        strcpy(out + j, s);
        j+=len;
    }
    return out;
}

```

2.2 מימוש רשימות מקושרות

מימוש בשביל לבדיק סטים

```c
#include "list.h"

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

int getListLength(Node list) {
    int c = 0;
    while (list != NULL) {
        c++;
        list = list->next;
    }

    return c;
}

bool isListSorted(Node list) {
    while (list != NULL && list->next != NULL) {
        if (!(list->x <= list->next->x)) {
            return false;
        }
        list = list->next;
    }
    return true;
}

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

ErrorCode insert(int x, Node *head) {
    Node newNode = createNode(x);
    if (newNode == NULL) {
        destroyList(*head);
        head = NULL;
        return MEMORY_ERROR;
    }
    newNode->next = *head;
    *head = newNode;

    return SUCCESS;
}
```

```c
ErrorCode mergeSortedLists(Node list1, Node list2, Node *mergedOut) {
    *mergedOut = NULL;
    if (list1 == NULL || list2 == NULL) {
        return EMPTY_LIST;
    }
    if (isListSorted(list1) == false || isListSorted(list2) == false) {
        return UNSORTED_LIST;
    }
    while (list1 != NULL && list2 != NULL) {
        int lowestVal;
        if (list1->x < list2->x) {
            lowestVal = list1->x;
            list1 = list1->next;
        } else {
            lowestVal = list2->x;
            list2 = list2->next;
        }
        if (insert(lowestVal, mergedOut) == MEMORY_ERROR)
            return MEMORY_ERROR;
    }

    while (list1 != NULL) {
        if (insert(list1->x, mergedOut) == MEMORY_ERROR)
            return MEMORY_ERROR;

        list1 = list1-> next;
    }
    while (list2 != NULL) {
        if (insert(list2->x, mergedOut) == MEMORY_ERROR)
            return MEMORY_ERROR;
        list2 = list2->next;
    }

    return SUCCESS;
}
```