1.1.2 סעיף א

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

char *string[1]duplicator(char *s, inttimes)[2]{
    assert(!s);
    assert(times > 0);
    int [3]LEN= strlen(*s);
    char *out = malloc(LEN * times);
    assert(out);
    for (int i = 0; i < times; i++) {
    [4]out = out + LEN;
    [4]strcpy(out, s);
    }
    return out;
}
```

שגיאות הקונבנציה מסומנות בצהוב, וממוספרות לפי המקרא הבא

[1] מילה שאינה ראשונה בשם פונקציה צריכה להתחיל באות גדולה – stringDuplicator
[2] סוגר פותח של פונקציה צריך להיות בשורה חדשה
[3] שם משתנה צריך להיות באותיות קטנות - len במקום LEN
[4] בלוק של לולאה צריך להיות בהזחה

שגיאה נוספת – שם משתנה צריך להיות משמעותי, עבור char במקום s צריך להיות str

2.1.1 סעיף ב

הקוד המתוקן

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <assert.h>
5
6   char *stringDuplicator(char *str, int times)
7   {
8       assert(str);
9       assert(times>0);
10      int len=strlen(str);
11      char *out=(char*)malloc(sizeof(char)*(len*(times+1)));
12      if (out==NULL)
13      {
14          printf("Memory allocation failed");
15          exit(1);
16      }
17      char* temp=out;
18      for (int i=0; i<times; i++)
19      {
20          strcpy(temp, str);
21          temp+=len;
22      }
23      return out;
24  }
```

שגיאות התכנות מפורטות לפי מספר השורה שבה הן מופיעות בקוד המתוקן

8) assert מקבל את ההנחות שמתקיימות בחיוב, לכן הארגומנט צריך להיות (str) ולא (!str)

11) strlen נותנת את אורך המחרוזת לא כולל תו הסיום 0\ , ולכן כאשר נרצה להקצות דינאמית מקום למחרוזת חדשה באותו האורך יש להוסיף 1 עבור תו הסיום. (times +1) ולא (times)

12) עבור מאלוק יש לבדוק אם הצליח ולא להניח כי הצליח!, לכן assert לא מתאים כאן ויש לכתוב במקומו בדיקה להצלחת המאלוק

17) יש ליצור משתנה זמני שישמור את הערך של out מכיוון שבלולאה אנחנו מקדמים את out להצביע לסוף המחרוזת

20-21) יש להחליף בין השורות הנ"ל. קודם מעתיקים את המחרוזת אחרי זה מקדמים את המצביע

2.2מיזוג רשימות מקושרות ממוינות

```
ErrorCode mergeSortedLists(Node list1, Node list2, Node* mergedOut)
{
    // if either list is empty
    if (list1==NULL || list2==NULL)
    {
        mergedOut=NULL;
        return EMPTY_LIST;
    }
    // if either list is unsorted
    if (!isListSorted(list1) || !isListSorted(list2))
    {
        mergedOut=NULL;
        return UNSORTED_LIST;
    }
    // if mergedOut is NULL
    if (mergedOut==NULL)
    {
        mergedOut=NULL;
        return NULL_ARGUMENT;
    }
    Node dummy=createNode(0);
    if (dummy==NULL)
    {
        return MEMORY_ERROR;
    }
    Node tail=dummy;
    //classic merge:
    //as long as it's not the end of the lists
    while (list1!=NULL && list2!=NULL)
    {
        if (list1->x < list2->x)
        {
            Node new_node=createNode(list1->x);
            if (new_node==NULL)
            {
                destroyList(dummy);
                *mergedOut=NULL;
                return MEMORY_ERROR;
            }
            moveTail(&list1, &tail, &new_node);
        }
        else
        {
            Node new_node=createNode(list2->x);
            if (new_node==NULL)
            {
                destroyList(dummy);
                *mergedOut=NULL;
                return MEMORY_ERROR;
            }
            moveTail(&list2, &tail, &new_node);
        }
    }
    //if we here list2 ended
    ErrorCode result1=copyRestOfList(&list1, &tail, &dummy, mergedOut);

    //if we here list1 ended
    ErrorCode result2=copyRestOfList(&list2, &tail, &dummy, mergedOut);

    if (result1==MEMORY_ERROR || result2==MEMORY_ERROR)
    {
        free(dummy);
        return MEMORY_ERROR;
    }
    else {
        *mergedOut=dummy->next;
        free(dummy);
        return SUCCESS;
    }
}
```

פונקציות עזר שמימשנו:

```
ErrorCode copyRestOfList(Node* list, Node* tail, Node* list_dummy, Node* mergedOut)
{
    while ((*list)!=NULL)
    {
        Node new_node=createNode((*list)->x);
        if (new_node==NULL)
        {
            destroyList(*list_dummy);
            *mergedOut=NULL;
            return MEMORY_ERROR;
        }
        moveTail(list, tail, &new_node);
    }
    return SUCCESS;
}
```

```
void moveTail(Node* list, Node* tail, Node* new_node)
{
    (*tail)->next=*new_node;
    *tail=*new_node;
    *list=(*list)->next;
}
```

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

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