```c

/* ********************************************************************** */
/* *************** Dry part - PART 1 ***************************** */
/* ********************************************************************** */




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

Node nodeCreateOrDestroyAll(Node head, int data) {
    Node node = malloc(sizeof(*node));
    if (node == NULL) {
        listDestroy(head);
        return NULL;
    }
    node->x = data;
    node->next = NULL;
    return node;
}


Node listCopyOrDestroyAll(Node source, Node dest, Node dest_head)
{
    Node destination_ptr = dest;
    assert(source != NULL);
    while(source) {
        destination_ptr->next = nodeCreateOrDestroyAll(dest_head, source->x);
        if (destination_ptr->next == NULL) {
            return NULL;
        }
        destination_ptr = destination_ptr->next;
        source = source->next;
    }
    return dest;
}

ErrorCode mergeSortedLists(Node list1, Node list2, Node* merged_out) {
    // one of the lists is empty
    if (list1 == NULL || list2 == NULL) {
        return EMPTY_LIST;
    }
    Node merged_ptr = *merged_out;
    // check if lists are sorted
    if(!isListSorted(list1) || !isListSorted(list2)) {
        return UNSORTED_LIST;
    }
    //both lists are sorted and full
    while (list1 != NULL && list2 != NULL) {
        int merge_from = (list1->x <= list2->x) ? 1 : 2;
        switch (merge_from) {
            case 1:
                if(*merged_out == NULL) { // merged list is empty
                    *merged_out = nodeCreateOrDestroyAll(*merged_out, list1->x);
                    if(*merged_out == NULL){
                        return MEMORY_ERROR;
                    }
                    merged_ptr=*merged_out;
                }
                else {      // merged list is not empty
                    merged_ptr->next = nodeCreateOrDestroyAll(*merged_out, list1->x);
                    if (merged_ptr->next == NULL) {
                        return MEMORY_ERROR;
                    }
                    merged_ptr = merged_ptr->next;
                }
                list1 = list1->next;
                break;
            case 2:
                if(*merged_out == NULL) { // merged list is empty
```

```
134                     *merged_out = nodeCreateOrDestroyAll(*merged_out, list2->x);
135                     if(*merged_out == NULL){
136                         return MEMORY_ERROR;
137                     }
138                     merged_ptr=*merged_out;
139                 }
140                 else {      // merged list is not empty
141                     merged_ptr->next = nodeCreateOrDestroyAll(*merged_out, list2->x);
142                     if (merged_ptr->next == NULL) {
143                         return MEMORY_ERROR;
144                     }
145                     merged_ptr = merged_ptr->next;
146                 }
147                 list2 = list2->next;
148                 break;
149         }
150     }
151     Node rest_of_list = NULL;
152     // finished with one of the lists
153     if (list1 == NULL) {    // finished with list1
154         rest_of_list = listCopyOrDestroyAll(list2, merged_ptr, *merged_out);
155         if (rest_of_list == NULL) {
156             return MEMORY_ERROR;
157         }
158     }
159     else {  // finished with list2
160         rest_of_list = listCopyOrDestroyAll(list1, merged_ptr, *merged_out);
161         if (rest_of_list == NULL) {
162             return MEMORY_ERROR;
163         }
164     }
165     return SUCCESS;
166 }
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
```

```c

/* ************************************************************** */
/* *************** Dry part - PART 2 ***************************** */
/* ************************************************************** */


char *stringDuplicator(char *s, int times) {
    // PROGRAMMING ERROR 1: s should be transferred to the function as const to
    // CONVENTION ERROR 1: s name should be src
    // CONVENTION ERROR 2: func name should be a verb
    assert(!s);
    // PROGRAMMING ERROR 2: should be assert(s)
    assert(times > 0);
    int LEN = strlen(s);
    // CONVENTION ERROR 3: LEN - variable names should be in lower case
    // PROGRAMMING ERROR 3:we should allocate an additional bye for the /0  (strlen() returns the length of the string without the /0 char)
    char *out = malloc(LEN * times);
    assert(out);
    // PROGRAMMING ERROR 4: should be if(!out) { return NULL; }
    for (int i = 0; i < times; i++) {
        // CONVENTION ERROR 4: no indent lines in for loop
        out = out + LEN;
        // PROGRAMMING ERROR 5: this two lines should flip - first copy, than increment pointer
        strcpy(out, s);
    }
    return out;
    // PROGRAMMING ERROR 6: returning a pointer to the end of out string
}


// **********  Fixed: **********


char* stringDuplicate(const char *src, int times) {
    assert(src);
    assert(times > 0);
    char* out = malloc(strlen(src) * (times + 1));
    if(!out) {
        return NULL;
    }
    for (int i = 0; i < times; i++) {
        strcat(out, src);
    }
    return out;
}
```