Merge Sorted Nodes:

```
void nodeDestroy(Node to_destroy)
{
    if(to_destroy == NULL)
    {
        return;
    }
    Node tmp;
    while(to_destroy != NULL)
    {
        tmp = to_destroy;
        to_destroy = to_destroy->next;
        free(tmp);
    }
}

ErrorCode mergeSortedLists(Node list1, Node list2, Node *merged_out)
{
    if(list1 == NULL || list2 == NULL)
    {
        return EMPTY_LIST;
    }
    if(!isListSorted(list1) || !(isListSorted(list2)))
    {
        return UNSORTED_LIST;
    }
    if(merged_out == NULL)
    {
        return NULL_ARGUMENT;
    }
    Node* starter_merged = merged_out;
    int len1 = getListLength(list1);
    int len2 = getListLength(list2);
    while(len1 > 0 && len2 > 0)
    {
        assert(list1 != NULL && list2 != NULL);
        (*merged_out)->next = malloc(sizeof(*(*merged_out)));
        if((*merged_out)->next == NULL)
        {
            nodeDestroy(*starter_merged);
            return MEMORY_ERROR;
        }
        if(list1->x >= list2->x)
        {
            (*merged_out)->x = list2->x;
            len2--;
            list2 = list2->next;
            merged_out = &((*merged_out)->next);
        }
```

```
        else if(list1->x < list2->x)
        {
            (*merged_out)->x = list1->x;
            len1--;
            list1 = list1->next;
            merged_out = &((*merged_out)->next);
        }
    }
    while(len1 > 0)
    {
            assert(list1 != NULL);
            (*merged_out)->x = list1->x;
            len1--;
            list1 = list1->next;
            if(len1 > 0)
            {
                (*merged_out)->next = malloc(sizeof(*(*merged_out)));
                if((*merged_out)->next == NULL)
                {
                    nodeDestroy(*starter_merged);
                    return MEMORY_ERROR;
                }
                merged_out = &((*merged_out)->next);
            }
    }
    while(len2 > 0)
    {
        assert(list2 != NULL);
        (*merged_out)->x = list2->x;
        len2--;
        list2 = list2->next;
        if(len2 > 0)
        {
            (*merged_out)->next = malloc(sizeof(*(*merged_out)));
            if((*merged_out)->next == NULL)
            {
                nodeDestroy(*starter_merged);
                return MEMORY_ERROR;
            }
            merged_out = &((*merged_out)->next);
        }
    }
    (*merged_out)->next = NULL;
    return SUCCESS;
}
```

Convention & Coding errors:

```
char *stringDuplicator(char *s, int times) { //Convention error - unconventional "s" shortcut
                                                for string/str
                                    //Convention error - bad function name - it is not a verb.
                                    //Coding error - s must be of type const char*
    assert(!s);
    assert(times > 0);
    int LEN = strlen(s); //Convention error - variable name is in capital letters - "LEN"
    char *out = malloc(LEN * times);
    assert(out);
       //Coding error - memory allocation must be checked during runtime and not as an assert
    for (int i = 0; i < times; i++) {
    out = out + LEN; //Coding error - does not start at the initial 0th index, leaving behind
                                       empty memory cells
    strcpy(out, s); //Coding error - at the last loop, will encounter a segmentation error
                      for leaving the bounds of the malloc
                    //Convention error - no indentation!
    }
    return out; //Coding error - returns wrong address to string, only returns the last
                  duplication (outside the malloc allocation...)
}
```

Fixed code:

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