Question 1: You must be kidding! (groan) (15 pts, 40 min)
We have a simple linked list that consists of kids’ names (a standard C string) and the grade they are in – an integer between 0 (Kindergarten) and 12. The structure appears as follows, with an example:

```c
typedef struct kid_node {
    int grade;
    char *name;
    struct kid_node *next;
} kid_t;
```

For “administrative reasons”, we’d like to categorize our kids by grade. We copy the kids’ information into an array of linked lists indexed by the grade.

```
#define MAXGRADE 12

```
b) For every Yin, there is a Yang. Now that we have a function for creating kid arrays, we must create a function that frees all memory associated with the structure. Fill in the following functions. free_kid_array calls the recursive function free_kid_list which frees a single kid list.

```c
void free_kid_array(kid_t *kid_array[]) {
    int i;
    for (i = 0; i <= MAXGRADE; i++) {
        free_kid_list(kid_array[i]);
    }
    /* Clean up if necessary */
}

void free_kid_list(kid_t *kid) {
    /* Declare temp variables */
    if (kid == NULL)
        return;
}
```
Question 2: C (16 Points – 30 minutes)

We’ve written matchSubStr below in C. Some of the lines are buggy and some are perfectly fine. Circle BUGGY or OK for each labeled C statement and if buggy, why it is and provide the fix. A line may be buggy for multiple reasons, so be sure you're descriptive.

Use the comments near each statement as a guide for what the line SHOULD do. If the code is buggy and you have a more clever/intuitive way of doing the same thing, feel free to do it your way.

Note: You can assume only valid input will be provided (two non-empty, null-terminated strings).

/* This function tries to find a substring (sub) within another (string). * If matchSubStr() finds the substring, it returns the index of the start * of the substring. If there is more than one match, it returns the first. * This is the scheme-equivalent of an equal? match (not eq? match) */
int matchSubStr(char sub[], char string[]) {

    /* Holds the location we’re checking (and will return if a match). */
    A:  int loc;          || BUGGY If buggy, why?      
        || OK If buggy, fix:

    /* These variables are pointers to the chars in sub/string */
    B:  char *c1, c2;      || BUGGY If buggy, why?      
        || OK If buggy, fix:

    /* We want to iterate through the string looking for a match, so we start at * loc=0 (beginning) and keep going as long as we have characters remaining */
    C:  for(loc=0; strlen(string[loc]); loc++) { || BUGGY If buggy, why?      
                                                        || OK If buggy, fix:

    /* We step through the substring using c1 and c2 to reference the * letters in sub and string. We stop when we have either exhausted * all the characters in sub (and thus found a match) or when we * encounter two characters that are not equivalent. */
    D:  for(c1 = sub, c2 = string&loc;              || BUGGY If buggy, why?      
                                      || OK If buggy, fix:
    E:    ;                                       || BUGGY If buggy, why?      
                                      || OK If buggy, fix:
    F:    c1++, c2++) {                            || BUGGY If buggy, why?      
                                                        || OK If buggy, fix:

    /* If we didn’t find a match, we break out */
    G:    if(c1 != c2) { || BUGGY If buggy, why?      
                                 || OK If buggy, fix:
        break;
    }

    /* We return the location if we found a match */
    H:    if(loc) {
        return loc;
    }
    }

    /* Return -1 if we didn’t find a match */
    return -1;
}
**Question 3:** Numerical Representation (10 points – 20 min.)
Considering 8-bit integers, answer the following questions for each column. The bits are numbered as: 7 6 5 4 3 2 1 0. Each box might be a different integer. You must show scratch work to receive credit.

<table>
<thead>
<tr>
<th>Sign-magnitude</th>
<th>Given that bits 3-0 are 1111</th>
<th>Given that bits 7-4 are 1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>- CAN'T-TELL +</td>
<td>- CAN'T-TELL +</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scratch space</th>
<th>Scratch space</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Given that bits 3-0 are 1111</th>
<th>Given that bits 7-4 are 1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scratch space</td>
<td>Scratch space</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scratch space</th>
<th>Scratch space</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Two's complement signed</th>
<th>Given that bits 3-0 are 1111</th>
<th>Given that bits 7-4 are 1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scratch space</td>
<td>Scratch space</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scratch space</th>
<th>Scratch space</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>One's complement</th>
<th>Given that bits 3-0 are 1111</th>
<th>Given that bits 7-4 are 1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scratch space</td>
<td>Scratch space</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scratch space</th>
<th>Scratch space</th>
</tr>
</thead>
</table>