These questions were provided courtesy of Vinay Nagar (with some minor tweaking by Steve Lumetta).

ECE198KL Midterm #3 Sample Questions

Linked Lists:

1. Intersection point of two linked lists (Simple)

In a database of a certain university, students enrolled in a particular course are maintained as linked lists with each node representing. The student structure is defined as follows:

```
struct student_node {
        char* name; /* name of the student */
        struct student_node* next; /* pointer to the next node in the list  */
};
```

If two courses have overlap, we can save a little memory by reusing some of the student structures (the lists can merge and share a common sublist of nodes).

Given two such linked list representing students enrolled for two different courses, say ECE 198KL and ECE 101, determine whether the lists do merge, and, if so, identify the first student in both classes (the head of the shared sublist). (For a challenge, do so in O(N lg N) steps rather than O(N^2) steps for a total number of nodes N.)

Print the names of the students if any, each in a new line. You should not print anything if there is no common student.

---

2. Rotate alternate 'k' nodes in a linked list (Medium, recursion based ideally)

A linked list where a node is represented as

```
struct node {
        int data;
        struct node *next;
};
```

can be reversed as follows:

```
struct node* result = NULL;
struct node* current = head;
struct node* next;
while (current != NULL) {
        next = current->next;
        current->next = result;
        result = current;
        current = next;
}
*newhead = result;
```

Given this piece of code, reverse every alternate 'k' nodes in the linked list and return the new head pointer. You can assume that the number of nodes in the linked list will always be a multiple of 'k'.

Eg:
Input: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9, k = 3
Output: 3 → 2 → 1 → 4 → 5 → 6 → 9 → 8 → 7

Your recursion should operate on groups of k or 2k nodes, not on individual nodes of the list (you can write a recursive function on individual nodes, but it will be substantially more complicated).

Recursion

1. In a binary tree data structure with “left” and “right” children, an ancestor of a node is defined as any node that is part of the path from the root to the node. (Hard)

Given two nodes node1, node2, in a tree, identify the first common ancestor of the nodes.

Hint: You can follow a path where node1 and node2 are both on the same side. Eg: if both nodes are in the right branch, take that branch. With this method, when node1 and node2 are no longer on the same side, then you should have found the first common ancestor.

Implement your logic in the following functions

```
struct node* commonAncestor (struct node* root, struct node* node1, struct node* node2);
bool isPath (struct node root, struct node* node);  /* checks if there is a path from root to node
*/
```

---

2. Addition of numbers represented as linked list (Medium)

Two decimal numbers are represented as linked list in reverse order where each node contains a single digit. Implement the function struct node* addLinkedLists(struct node* list1, struct node* list2) that adds the two numbers and returns the result. Create your own recursive function (call it once from the function addLinkedLists) so that you can pass any additional information from call to call.

You can assume that the size of the result linked list will be at least as large as the size of the larger of the two linked lists.

Eg:
list1: 1 → 2 → 3 (actual number, 321)
list2: 4 → 9 → 2 (actual number, 294)
Result: 5 → 1 → 6 (actual number, 615)

Eg:
list1: 1 → 2 (actual number, 21)
list2: 4 → 5 → 6 (actual number, 654)
Result: 5 → 7 → 6 (actual number 675)

---

3. Sorted merge of two linked list (Easy)

Sorted merge is an essential part of a sorting algorithm called "Merge Sort". The idea is to merge lists that are sorted into one list. Given two linked lists where each node is represented as:

```
struct node {
        int data;
        struct node *next;
};
```

implement the sorted merge algorithm in the function struct node* sortedMerge (struct node* list1, struct node* list2) . The resultant linked list will have all the nodes from both the linked lists in sorted order. Note that all the nodes should be added to the list that has the smallest element. (list1 in the example below). No new linked list should be created. You may assume that there will not be repeated nodes in the linked lists.

Input:
list1: 3 → 5 → 10 → 21
list2: 4 → 6 → 15 → 26 → 33
Output:
3 → 4 → 5 → 6 → 10 → 15 → 21 → 26 → 33

Dynamic allocation:

1. To keep track of the students' scores in a course, a TA creates a structure and does some initialization as follows: (Simple)

```
typedef struct student {
        int SID;  // Student ID (>= 1, <= 150) */

        char name[20]; // Name of the student. (20 characters or fewer)
        int num_tests; // Number of tests the student has taken (>= 1
                        // and <= 10)
        int *tests; /* Pointer to the beginning of a dynamically
                        allocated array, each location
                        representing test score between 1 and 100 */

} ;
struct student *records = NULL;  /* A global pointer to the beginning
                        of the dynamically allocated array of type student */
```

In the program file given to you, implement the following functions. You should allocate memory on heap using standard library function calls like malloc.

void allocateRecords (int num_students) → allocate 'num_students' records and make the global pointer 'records' point to the newly allocated memory. Note that 'records' should point to NULL if memory allocation fails.

void allocateTests(int num_tests) - Allocate memory for 'num_tests'
for each of the records

void freeMemory (int num_students) – Free all the allocated memory

For fun, check out the tools on EWS: Run a Valgrind command to check whether all the memory allocated was freed. You should see a statement "All heap blocks were freed -- no leaks are possible" in the output when you run the script.

---

2. Short answer

Given an double pointer, int **array_pointer, allocate memory using dynamic allocation library calls such that the 'array_pointer' should represent a 2D array with 'ROWS' rows and 'COLS' columns and each element in the array should be accessible as array_pointer[i][j].