**Name:** Ito Choi **ID:** 1801787.

**Problem 1:**

a. Implement a recursive function sumOfDigits that takes a positive integer as input and returns the sum of all of the digits in the integer. For example, the sum of 55 is 10, the sum of 123 is 6, etc. Do not use any global or static variables.

```
int sumOfDigits(int num) {
    int mostSignificantDigits = num / 10;
    int leastSignificantDigits = num % 10;

    if (mostSignificantDigits == 0)
        return num;

    return leastSignificantDigit + sumofDigits (mostSignificant Digits)

}
```

b. How many times is sumOfDigits called for any given number? Use induction: how many for the number 1? how many for the number 432? how many for 5555? etc.

→ We're reducing num by one digit each recursive call until we reach only one digit, So numOfDigits will be called on the order of the number of digits in num.

**Name:** Ithe Choi **ID:** 180n87.

**Problem 2:**

a. A palindrome is a word, phrase or sequence that reads the same backward as forwards, for example, "bob", "step on no pets". Write a recursive function isPalindrome that takes a cstring and its size as input and returns true if it is a palindrome, false otherwise. The size must be the same as the number of actual characters in the cstring. Do not use any global or static variables. Below is some example usage:

```
isPalindrome("tacocat", 7); // true
isPalindrome("abka", 4);    // false
```

You should not require any additional libraries or functions to implement this function.

```
bool isPalindrome(const char * s, int size) {
    if (size <=1)
        return true;

    if (s[0] == s[size-1])
        return isPalindrome(s+1, size-2);

    else
        return false;

}
```

b. How many times if isPalindrome called for any given cstring? Use induction, how many for "a"? how many for "aa"? how many for "aba"? etc.

→ Given any cstring we're reducing its size by two characters until we reach one, if odd number of characters, or zero, if even number of characters. Let n be the number of characters in isPalindrome, we'll call isPalindrome roughly ceil(n/2) times.

**Name:** Inho Choi **ID:** 1801178.

**Problem 3**

a. Implement a recursive function deleteList, that takes pointer to the head of a singly linked list and deletes the whole list.

```
struct Node {
        int val;
        Node* next;
};

void deleteList(Node* head) {
        if (head == nullptr)
          return;
        deleteList(head->next);
        delete head.
```

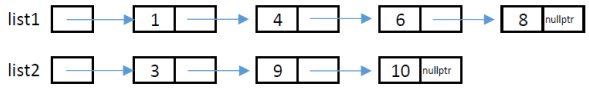
b. How many calls to deleteList would you make if you passed in a linked list with q nodes?

→ deleteList will be called q times plus 1 for nullptr.

**Name:** Isho Choi **ID:** 180178?

**Problem 4:**

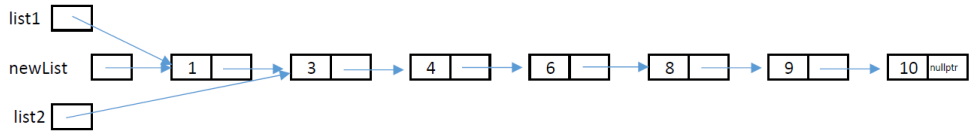
a. Given the Node definition above, implement a recursive function that merges two sorted singly linked lists into a single sorted linked list. The function should return the head of the new list. You may not create any new Nodes, this is known as an in-place merge. For example suppose we have two list:




And we call our merge function on these two lists:

```
Node* newList = inPlaceMerge(list1, list2);
```

The state of our program after that function call may look like:




```
Node* inPlaceMerge(Node* list1, Node* list2) {
if(list1 == nullptr) return list2;
if(list2 == nullptr) return list1;

Node *smallestNode = nullptr, *mergedSubList = nullptr;
if(list1->val < list2->val) {
        smallestNode = list1;
        mergedSubList = inPlaceMerge(list1->next, list2);
} else {
    smallestNode = list2;
    mergedSubList = inPlaceMerge(list1, list2->next);
}
smallestNode->next = mergedSubList;
return smallestNode;
}
```

b. If list1 has **n** nodes and list2 has **m** nodes, roughly how many times is inPlaceMerge called? Use induction: if list1 has 1 node and list2 has 4 nodes? If list1 has 3 and list2 has 2 nodes?

We're reducing the size of the problem by one node from either list until one of the lists is empty. At that point we know that the remainder of the merging will be from the nonempty list so we can just tack on those nodes to what has already been merged. So we'll stop calling inPlaceMerge once one of the lists runs out of nodes.
There can be any number of reasonable combos depending on the relative values
In either list one list is empty to begin with, then we only call inPlaceMerge

once. Otherwise, assuming neither lists are empty and if we assume that the lists are constructed in such a way that, on average, we go back and forth between lists grabbing nodes e.g. one list has consecutive evens and the other consecutive odds, then we'll incur that maximum number of calls to inPlaceMerge. In this case it'll be roughly 2 * min(m,n).



**Name:** **ID:**

**Extra space for inPlaceMerge if provided space is not sufficient:**

```
Node* inPlaceMerge(Node* list1, Node* list2) {

}
```