Problem-1:
Reverse a linked list using recursion

Test cases:

1. Positive case: Reverse({1, 21, 3, 45, 15}) ⇒ **| 15 | -> | 45 | -> | 3 | -> | 21 | -> | 1 | -> Nil**
2. Negative case: Reverse(NULL) ⇒ **Nil**
3. Boundary case: 1 element: Reverse({1}) ⇒ **| 1 | -> Nil**
4. Positive case 2: Reverse({1, 2, 3}) ⇒ **| 3 | -> | 2 | -> | 1 | -> Nil**
5. Boundary case: 2 elements: Reverse({1, 2}) ⇒ **| 2 | -> | 1 | -> Nil**
6. Positive case, longer list: Reverse({1, 11, 45, 12, 67, 89, 91}) ⇒ **| 91 | -> | 89 | -> | 67 | -> | 12 | -> | 45 | -> | 11 | -> | 1 | -> Nil**
7. Even# list: Reverse({{1, 11, 45, 12, 67, 89}) ⇒ **| 89 | -> | 67 | -> | 12 | -> | 45 | -> | 11 | -> | 1 | -> Nil**

Solution:

```
typedef struct _node {
   struct _node *next;
   int data;
} Node;

Node* reverse(Node *tail, Node *prev)
{
   if (!tail)
   {
      // If we've reached the end of the list,  prev will be the new head
      return prev;
   }

   // Recursively reverse the tail
   Node *head = reverse(tail->next, tail);

   // Tail's next link should now point to prev
   tail->next = prev;

   // prev's next link should be null
   if ( prev )
   {
      prev->next = NULL;
   }

   return head;
}
```

Problem-2a:
Alternative node split: Given a linked list, split it into two such that every other node goes into the new list. For lists with odd number of nodes, first one should be longer. For example: an input list: {a, b, c, d, e, f, g} results in {a, c, e, g} and {b, d, f}.

Test cases:

- alternativeListSplit({1, 2, 3, 4, 5}) ⇒ {1, 3, 5} and {2, 4}
- alternativeListSplit({47, 32, 12, 1}) ⇒ {47, 12} and {32, 1}
- alternativeListSplit({1}) ⇒ {1}, NULL
- alternativeListSplit(NULL) ⇒ NULL, NULL

```
Node* alternativeListSplit(Node* list)
{
   if(!list)
   {
      return NULL;
   }

   Node* n1 = list;
   Node* head2 = n1->next;
   Node* n2 = head2;

   while(n1 && n1->next)
   {
      n1->next = n1->next->next;
      if(n2->next)
      {
         n2->next = n2->next->next;
      }
      n1 = n1->next;
      n2 = n2->next;
   }

   if (n2)
   {
      n2->next = NULL;
   }
   return head2;
}
```

Problem-2b:
Find the middle element of a linked list in one pass.

Test cases:

- Positive case: findMiddleNode({1, 21, 3, 45, 15}) ⇒ **3**
- Negative case: findMiddleNode(NULL) ⇒ **NULL**
- Boundary case: 1 element: findMiddleNode({1}) ⇒ **| 1 | -> 1**
- Positive case 2: findMiddleNode({1, 2, 3}) ⇒ **2**
- Boundary case: 2 elements: findMiddleNode({1, 2}) ⇒ **2**
- Positive case, longer list: findMiddleNode({1, 11, 45, 12, 67, 89, 91}) ⇒ **12**
- Even# list: findMiddleNode({{1, 11, 45, 12, 67, 89}) ⇒ **12**

Solution:

A typical solution involves computing the length of the list and then traversing exactly half of that. A faster approach is to use a slow pointer and a fast pointer, with the faster one traversing twice as fast. By the time the fast one reaches the end of the list, the slower one is at the mid-point. This is still O(n) but is twice as fast.

```
typedef struct _node {
   struct _node *next;
   int data;
} Node;

Node* findMiddleNode(Node* head)
{
   if (!head)
   {
      return head;
   }

   Node *slow_ptr = head;
   Node *fast_ptr = head;

   while(fast_ptr && slow_ptr)
   {
      // Fast_ptr is at the end of the linked list
      if (!fast_ptr || !fast_ptr->next)
      {
         break;
      }
      slow_ptr = slow_ptr->next;
      fast_ptr = fast_ptr->next->next;
   }
   return slow_ptr;
}
```

Problem-3:

Write a function that checks if the given input string has matching opening and closing parentheses. Valid parantheses are: '(', ')', '{', '}', '[', ']'

Test cases:

1. hasMatchingParantheses("((1 + 2) * 3)") ⇒ true
2. hasMatchingParantheses("({ 1 + 2) * 3)") ⇒ false
3. hasMatchingParantheses("(((1 + 2) * 3))") ⇒ true
4. hasMatchingParantheses("[([])}") ⇒ false
5. hasMatchingParantheses("} (1 * 2) + 3 * (5 - 6)") ⇒ false

Solution:

As the input string is scanned, determining if an opening brace has a corresponding closing one (and vice versa) gets complicated without remembering previous state. When a closing brace is encountered, the *most recent* brace should be a matching one, if not the expression is invalid. Stacks are an obvious choice for retrieving the most recent values. We can simply store opening braces as we see them, popping them after successfully verifying that they match the closing brace encountered. By the time we reach the end of the string, the stack should be empty.

Following is an implementation in O(n) time with O(n) space for the stack.

```
// Javascript Arrays provide the default interface for Stacks, but simple wrapper
// to hide underlying Array usage
function Stack() {
   this.stack = new Array();
   this.push = function (item) {
      this.stack.push(item);
   }
   this.pop = function () {
      return this.stack.pop();
   }
}

var openingBraceRegex = new RegExp('[{(\\[]');
var paranthesisRegex = new RegExp('[{}()\\[\\]]');
var matchingBraces = { "}" : "{", "]": "[", ")": "("};

function hasMatchingParantheses(input) {
   var stack = new Stack();
   var validExpr = true;
   for (var i in input) {
      var ch = input[i];
      // Ignore non-paranthesis chars
```

```
        if (ch.match(paranthesisRegex) === null) {
            continue;
        }
        if (ch.match(openingBraceRegex) !== null) {
            stack.push(ch);
            continue;
        }
        // At this point, ch has to be a closing paranthesis
        // Get the expected opening brace for this one and compare
        if (stack.pop() !== matchingBraces[ch])
        {
            validExpr = false;
            break;
        }
    }
    return validExpr;
}
```

Problem - 4:

Find the longest substring that has matching opening and closing parentheses. Example: if the input string is "(((((()))))", the answer is: "(())"

Test cases

- assert(maxLenMatchingParen("()")==2);
- assert(maxLenMatchingParen("(())") == 4);
- assert(maxLenMatchingParen("()(())") == 6);
- assert(maxLenMatchingParen("(((((")==0);
- assert(maxLenMatchingParen(")))")==0);
- assert(maxLenMatchingParen("())")==2);
- assert(maxLenMatchingParen("(()")==2);
- assert(maxLenMatchingParen("((((())((((")==4);

Solution

A brute force solution is to find all substrings and check if they have matching parens. This will be a solution with O(n*n*n) complexity (n*(n-1)/2) substrings with O(n) check for matching.

Another variation uses the fact that if a matching paranthesis isn't found, the prefix can no longer be part of the substring we are looking for. This will be O(n*n) solution where we find all matching substrings at position i.

Finally, if we end with a non-matching closing paranthesis *i*, we can just jump to the string starting after *i.* Following is code for this solution that finishes in O(n) timeframe, with O(n) space complexity, using a stack.

```
size_t maxLenMatchingParen(const std::string& s)
{
   size_t maxStart = 0;
   size_t maxLen = 0, prevStart = 0;
   std::stack<int> stack;
   size_t strLen = s.size();
   for(size_t i = 0; i < strLen; i++)
   {
      if (s[i] == '(')
     {
         stack.emplace(i);
      } else
      {
         if (stack.empty())
         {
            // We found a closing paranthesis
            // that doesn't have corresponding opening paranthesis
            // Advance to next char
            prevStart = i + 1;
         } else {
            // Matching parantheses found
            // Compute the length of matching paranthesis
            // 1. If stack is empty, compute length from (previous) start-position - full match
            // 2. Else, length of matching substring is however far i has gotten from
            //    top of the stack. (as an illustration, imagine how i and top would change
```

```
            //    for every matching paranthesis)
            stack.pop();
            size_t start = stack.empty() ? prevStart - 1 : stack.top();
            size_t size = i - start;
            if (size > maxLen)
            {
               maxStart = start + 1;
               maxLen = size;
            }
         }
      }
   }
   std::string matchingSubStr = s.substr(maxStart, maxLen);
   printf("%s", matchingSubStr.c_str());
   return maxLen;
}
```

Problem - 5:
Swap kth node from the beginning, with kth node from the end.

Test cases:

- swapNodes(list({11, 4, 6, 7, 1, -99, 0, 2}), 2) ⇒ **| 11 | -> | 0 | -> | 6 | -> | 7 | -> | 1 | -> | -99 | -> | 4 | -> | 2 | -> Nil**
- swapNodes(list({{11, 4, 2}), 2) ⇒ **| 11 | -> | 4 | -> | 2 | -> Nil**
- swapNodes(list({11, 4, 2, 1}), 2) ⇒ **| 11 | -> | 2 | -> | 4 | -> | 1 | -> Nil**
- swapNodes(list({11, 2, 4, 1}), 3) => **| 11 | -> | 4 | -> | 2 | -> | 1 | -> Nil**
- swapNodes(list({3, 2, 2, 0}, 1) ⇒ **| 0 | -> | 1 | -> | 2 | -> | 3 | -> Nil**
- swapNodes(list(NULL), 1) ⇒ **Nil**

Solution:

```
void swapNodes(Node **head, Node* prev1, Node* node1, Node* prev2, Node* node2)
{
   // prev1 can be null when first node is being swapped.
   if (!node1 || !node2)
   {
      return;
   }

   if(!prev1)
   {
      *head = node2;
   } else {
      prev1->next = node2;
   }

   // Allows symmetrical case i.e., k is > n-k
   if (!prev2)
   {
      *head = node1;
   } else {
      prev2->next = node1;
   }

   Node* temp = node1->next;
   node1->next = node2->next;
   node2->next = temp;
}

// Swaps the k-th node from the beginning with k-th node from end
void swapNodes(Node** head, int k)
{
   if (!*head || k <= 0)
   {
      return;
   }

   // We need to get pointers to:
   // 1. Nodes before kth node, and n-kth node
   // 2. kth node and n-kth node
   // And then, we need to swap the nodes
   // We will use prev1, node1 for kth node
   // and prev2, node2 for n-kth node
   Node* prev1 = NULL, *prev2 = NULL;
   Node* node1 = *head, *node2 = *head, *tmp = *head;
```

```
    // Determine Kth node from the beginning
    while( --k > 0 && tmp)
    {
        prev1 = tmp;
        tmp = tmp->next;
    }

    // At this point, tmp is at kth node
    node1 = tmp;

    // Now, to find n-kth node, we start moving node2 and tmp together until tmp reaches the end
    // At that point, node2 will be pointing at n-kth node
    while ( tmp && tmp->next )
    {
        prev2 = node2;
        node2 = node2->next;
        tmp = tmp->next;
    }

    swapNodes(head, prev1, node1, prev2, node2);
}
```

Problem - 6:

Sort a linked list using merge sort.

Test cases:

```
void testMergeSort()
{
    int array1[] = {11, 4, 6, 7, 1, -99, 0, 2};
```

```
    Node* l1 = initListFromArray(array1, 8);
    print(mergesort(l1));                    // | -99 | -> | 0 | -> | 1 | -> | 2 | -> | 4 | -> | 6 | -> | 7 | -> | 11 | -> Nil

    int array2[] = {11, 8, -1};
    Node* l2 = initListFromArray(array2, 3);
    print(mergesort(l2));                    // | -1 | -> | 8 | -> | 11 | -> Nil

    int array3[] = {1};
    Node* l3 = initListFromArray(array3, 1);
    print(mergesort(l3));                    // | 1 | -> Nil

    int array4[] = {1, 2, 3, 4, 5};
    Node* l4 = initListFromArray(array4, 5);
    print(mergesort(l4));                    // | 1 | -> | 2 | -> | 3 | -> | 4 | -> | 5 | -> Nil

    print(mergesort(NULL));                  // Nil
}
```

Solution:

This is an interesting problem that combines the knowledge of merge sort with the fun of manipulating linked list pointers.

It is easier to code a solution that allocates new lists and copies the elements over during merge. But, this will imply that we need O(n) extra space. A better solution is to directly move the nodes around based on their values (unless the question explicitly says that the new list should be a totally different one).

Following implementation performs the merge sort without additional O(n) storage.

```
// Determines the length of a linked list. Doesn't account for lists with cycles.
size_t length(Node *head)
{
    size_t len = 0;
    while (head)
    {
        head = head->next;
        len++;
    }
    return len;
}

Node* merge(Node* l1, Node* l2)
{
    Node* current1 = l1;
    Node* current2 = l2;
    Node* newHead = NULL;
    Node* prev = NULL;

    // Initialize newHead
    if ( (l1 && l2 && l1->data < l2->data)
        || (l1 && !l2))
    {
        newHead = l1;
```

```
        current1 = current1->next;
    } else if( (l1 && l2 && (l2->data < l1->data))
            || (l2 && !l1))
    {
        newHead = l2;
        current2 = current2->next;
    }

    prev = newHead;
    while( current1 && current2 )
    {
        if (current1->data < current2->data)
        {
            prev->next = current1;
            prev = current1;
            current1 = current1->next;
        }
        else
        {
            prev->next = current2;
            prev = current2;
            current2 = current2->next;
        }
    }

    while(current1)
    {
        prev->next = current1;
        prev = current1;
        current1 = current1->next;
    }

    while(current2)
    {
        prev->next = current2;
        prev = current2;
        current2 = current2->next;
    }

    return newHead;
}

Node* recSortList(Node* head, size_t len)
{
    // Base condition
    if(len==0 || len==1)
    {
        return head;
    }

    size_t halfLen = len/2;
    Node *tail = head;
    Node *prev = head;
    for(size_t i=0; i<halfLen && tail; i++)
    {
        prev = tail;
        tail = tail->next;
```

```
    }

    // Actually split the list into two
    prev->next = NULL;

    Node* l1 = recSortList(head, halfLen);     // Length is passed to avoid recomputation.
    Node* l2 = recSortList(tail, len-halfLen);
    return merge(l1, l2);
}

Node* mergesort(Node* head)
{
    return recSortList(head, length(head));
}
```