Zach Bagley

Data Structures and Algorithms Portfolio

Including solutions to LeetCode problems and other exercises.

Contents

LeetCode #234: Palindrome Linked List
Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

For this problem, I explored two suboptimal approaches before settling on an optimal solution. Below are all three of my approaches.

The problem supplied this helper function to build a singly-linked list:

```
function ListNode(val, next) {
    this.val = (val===undefined ? 0 : val)
    this.next = (next===undefined ? null : next)
}
```

My first solution (JavaScript):

```
var isPalindrome1 = function(head) {

    let inputArray = [];
    let currentNode = head;

    while (currentNode) {
        inputArray.push(currentNode.val);
        currentNode = currentNode.next;
    }

    let pal = true;
    for (let i = 0; i < Math.floor(inputArray.length / 2); i++) {
        if (inputArray[i] !== inputArray[inputArray.length - i - 1]) {
            pal = false;
            break;
        }
    }

    return pal;
};
```

Notes:
The problem's goal is more simply accomplished with an array than with a linked list—we can simply iterate through the first half of the array from the front and the second half from the end simultaneously, testing each value from the front against the corresponding value from the end.

This code gives us the right answers (see Appendix A). However, it is far from optimal. The function iterates through the entire linked list to build the array—an O(n) time operation—and then iterates through half of the resulting array to compare to the other half—also O(n) time. Ultimately, the function's time complexity is O(n). Where we come up short is space complexity: building the array requires an additional data point for each data point in the input, resulting in O(n) space complexity. Building an array also removes all the advantages of using linked lists in the first place: the computer must now be able to come up with a series of uninterrupted adjacent spaces in memory to store the array.

My second solution (JavaScript):

```
var isPalindrome2 = function(head) {
    function DoubleNode(val, next, prev) {
        this.val = (val===undefined ? 0 : val)
        this.next = (next===undefined ? null : next)
        this.prev = (prev===undefined ? null : prev)
    }

    let newHead = new DoubleNode(head.val);
    let currentOld = head.next;
    let currentNew = newHead;

    while (currentOld !== null) {
        let newNode = new DoubleNode(currentOld.val);
        currentNew.next = newNode;
        newNode.prev = currentNew;
        currentNew = newNode;
        currentOld = currentOld.next;
    }

    let tail = currentNew;
    let left = newHead;
    let right = tail;
    let pal = true;

    while (left !== null && right !== null && left !== right && left.prev !== right) {
        if (left.val !== right.val) {
            pal = false;
            break;
        }
        left = left.next;
        right = right.prev;
    }

    return pal;
};
```

Notes:

I maintained the advantages of linked lists over arrays while gaining the ability to iterate through the data from both ends by converting the input singly linked list into a doubly linked list. I needed to build a class of doubly linked nodes, modifying LeetCode's provided ListNode() function into a DoubleNode().

This function simply adds a "prev" pointer to each node, pointing to the previous node in the list. I then use this function to build a doubly linked list from the provided singly linked list, keeping track of our new list's head and tail. Finally, I iterate through the list from

both its head and its tail, testing its front values against its end values until the pointers meet in the middle.

This approach maintains the advantages of using linked lists, but in the end, its space and time complexities are still both O(n). It must build an entirely new set of data from the given data for our doubly linked list, and iterate through all the data once to build the new list and again through half the data to test for palindromicity.

My third (optimal) solution (JavaScript):

```
var isPalindrome3 = function(head) {
    let tortoise = head;
    let hare = head;
    let pal = true;

    while (hare && hare.next) {
        tortoise = tortoise.next;
        hare = hare.next.next;
    }

    let prev = null;
    let current = tortoise;
    let nextTemp = null;
    while (current) {
        nextTemp = current.next;
        current.next = prev;
        prev = current;
        current = nextTemp;
    }

    let p1 = head;
    let p2 = prev;
    while (p2) {
        if (p1.val !== p2.val) pal = false;
        p1 = p1.next;
        p2 = p2.next;
    }

    return pal;
};
```

Notes:

The algorithm would not be able to complete the task with O(1) space complexity unless it left the input as it is given: in a singly linked list. But in a singly linked list, each node only contains a pointer to the next node, making it impossible to iterate from each end. If we could work out a way to reverse the linked list, we could test the original and reversed versions against each other; however, to store both an original and reversed copy of the linked list, we would end up using O(n) space again. We could limit this to O(1) space if we

could split the linked list in half, reverse one half (with no need to store the original version of the reverse half), and test each half against the other. Reversing the order of a linked list is an O(n) operation and is not incredibly complex. Iterating through each node, we store its original "next" pointer in a temporary variable, change the pointer to point to a "previous" or "prev" value (which is null at the beginning of the operation, as the head of the original list becomes the tail of the reverse list). After changing the pointer, prev is set to the current node and we use the temporary variable to the next node to iterate forward in the linked list.

Our challenge is now to find the middle point, where we can start reversing the second half of the linked list while leaving the first half unchanged. To accomplish this, we use Floyd's Cycle Finding Algorithm. The algorithm uses two pointers to iterate through the list: a "tortoise" pointer that iterates through one node per iteration, and a "hare" pointer that jumps two nodes per iteration. With the hare moving twice as fast as the tortoise, it will eventually come up behind the tortoise if there is a loop in the list. If the hare and tortoise find themselves on the same node after beginning their iterations, there is a cycle.

We are not trying to detect a cycle, but the idea of a slow-moving tortoise pointer and a fast-moving hare pointer is useful to us. If the hare moves twice as fast as the tortoise, it will reach the end of our list when the tortoise is at the mid-point of our list. That's how we find our mid-point: when the hare reaches the end, the tortoise's node is our middle node, and we can reverse the second half of the linked list beginning at the tortoise's position, being sure to save our final modified node as the head of the newly reversed half.

Finally, it's a simple matter of comparing the first half to the reversed second half by iterating again through the first half from its head and the second half from its new head.

This solution iterates through the input data a number of times to traverse, reverse, and compare the linked list; however, that number increases proportionally to the input, not exponentially, so we maintain O(n) time complexity. Where we have really improved is space complexity: we declare a fixed number of variables and only modify—never copy—the input data, giving us O(1) space complexity

LeetCode #100: Same Tree

Given the roots of two binary trees p and q, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

My solution (JavaScript):

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
```

```
/**
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {boolean}
 */
var isSameTree = function(p, q) {
    if (!p && !q) {
        return true;
    } else if (!p || !q) {
        return false
    }

    if (p.val !== q.val) {
        return false
    }

    return (isSameTree(p.left, q.left) && isSameTree(p.right, q.right))
};
```

Notes:
This algorithm uses recursion to traverse every possible route to every leaf node on both trees simultaneously, comparing the values of each node as it goes. If both nodes (p and q) are null, the trees are identical at that branch, so it returns true. If one node is null and the other is not, the trees differ, so it returns false. If both nodes exist, it compares their values. If they differ, the trees are not the same and it returns false. If the values are equal, it recursively checks the left subtrees (p.left and q.left) and the right subtrees (p.right and q.right). Both must be identical for the overall trees to be considered the same.

The algorithm's time complexity is O(N)—each node is visited exactly once in the worst case. Space complexity is not immediately intuitive but should likely be considered O(h). The space needed is proportional to the height of the tree (h), which is the recursion stack depth.

LeetCode #101: Symmetric Tree
Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

My solution:

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
```

```
/**
 * @param {TreeNode} root
 * @return {boolean}
 */

function isMirror(p, q) {
    if (!p && !q) {
    return true;
    } else if (!p || !q) {
        return false
    }

    if (p.val !== q.val) {
        return false
    }

    return (isMirror(p.left, q.right) && isMirror(p.right, q.left))
};

var isSymmetric = function(root) {
    if (!root) return true;
    return isMirror(root.left, root.right);
};
```

Notes:
My solution defines a helper function that is very similar to the isSameTree() function from the previous problem, except that it test whether the tree mirror each other (rather than if they are the same) by comparing each node's left child in the first tree to the right child in the second tree and vice versa.

The main function, isSymmetric(), tests if the root node is empty, then passes the roots of the tree's right subtree and left subtree to the isMirror() helper function.

Time complexity is O(n). Each node of the tree is visited once by the recursive function. In the worst-case scenario, all n nodes are checked.

Space complexity is O(h). The recursion stack depth is proportional to the height, h, of the tree. In the worst-case (a skewed tree), this can be O(n).

Binary Tree Visualizer (Not from LeetCode)
Prompt:
Step 1. Learn how to do a level-by-level traversal of a binary tree.
Step 2. Using the code from step 1 render your binary tree structure in the following style. Center the text for each node value in a 3-character field (padded with spaces as appropriate). If the value is empty because it's one of the empty nodes we added, just write out three spaces.
The following should all be true:

- *3 characters per field at the leaf level, with one space separating each field.*

- *Let h be the height of a node (a leaf has height 1), then the spacing between 3-character fields is 2^(h + 1) - 3.*
- *The spacing before the first field is 2^h - 2.*

Step 3. Now add the UTF-8 box-drawing characters in between the lines showing the node values. I call this "connector text". Here are a couple of things that should be true about connector text:

- *The number of horizontal line characters (–) between corner and T characters is 2^h - 1.*
- *The number of spaces before the first UTF-8 box-drawing character is also 2^h - 1.*

Write code sufficient to test your implementation.

My solution:

```
#!/usr/bin/env node

class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }
}

class BinaryTree {
  constructor() {
    this.root = null;
  }

  insert(value) {
    const newNode = new TreeNode(value);
    if (this.root === null) {
      this.root = newNode;
    } else {
      this.insertNode(this.root, newNode);
    }
  }

  insertNode(node, newNode) {
    if (newNode.value < node.value) {
      if (node.left === null) {
        node.left = newNode;
      } else {
        this.insertNode(node.left, newNode);
      }
    } else {
      if (node.right === null) {
        node.right = newNode;
      } else {
        this.insertNode(node.right, newNode);
      }
    }
  }

  delete(value) {
    this.root = this.deleteNode(this.root, value);
  }

  deleteNode(node, value) {
```

```
    if (node === null) {
      return null;
    } else if (value < node.value) {
      node.left = this.deleteNode(node.left, value);
      return node;
    } else if (value > node.value) {
      node.right = this.deleteNode(node.right, value);
      return node;
    } else {
      if (node.left === null && node.right === null) {
        node = null;
        return node;
      }
      if (node.left === null) {
        node = node.right;
        return node;
      } else if (node.right === null) {
        node = node.left;
        return node;
      }
      const aux = this.findMinNode(node.right);
      node.value = aux.value;
      node.right = this.deleteNode(node.right, aux.value);
      return node;
    }
  }

  search(node, value) {
    if (node === null) {
      return null;
    } else if (value < node.value) {
      return this.search(node.left, value);
    } else if (value > node.value) {
      return this.search(node.right, value);
    } else {
      return node;
    }
  }
}

function getMaxHeight(root) {
  if (root === null) {
    return 0;
  }
  return 1 + Math.max(getMaxHeight(root.left), getMaxHeight(root.right));
}

function convertToCompleteBinaryTree(root) {
  if (root) {
    fillTree(root, 1, getMaxHeight(root));
  }

  return root;
}

function fillTree(node, currentDepth, maxHeight) {
  if (node) {
    if (currentDepth < maxHeight) {
      if (!node.left) {
        node.left = new TreeNode("");
      }
```

```
      if (!node.right) {
        node.right = new TreeNode("");
      }

      fillTree(node.left, currentDepth + 1, maxHeight);
      fillTree(node.right, currentDepth + 1, maxHeight);
    }
  }
}

function getLevel(root, level, nodes = []) {
  if (!root) return nodes;
  if (level === 0) {
    nodes.push(root);
  } else {
    getLevel(root.left, level - 1, nodes);
    getLevel(root.right, level - 1, nodes);
  }
  return nodes;
}

function getAllLevels(root) {
  convertToCompleteBinaryTree(root);
  const height = getMaxHeight(root);
  let levels = [];
  for (let level = 0; level < height; level++) {
    levels.push(getLevel(root, level, []));
  }
  return levels;
}

function formatValue(val) {
  let str = String(val);
  if (str.length > 3) {
    str = str.slice(0, 3);
  }

  if (str.length == 0) {
    str = "   ";
  } else if (str.length == 1) {
    str = " " + str + " ";
  } else if (str.length == 2) {
    str = " " + str;
  }
  // while (str.length < 3) {
  //   str = str + "_";
  // }
  return str;
}

function prettyPrint(root) {
  console.log("Step 2 -- prettyPrint():")
  console.log("\n");
  let tree = getAllLevels(root);
  let height = getMaxHeight(root);
  for (let level in tree) {
    let levelStr = "";
    for (let i = 0; i < ((2 ** height) - 2); i++) {
      levelStr = levelStr + " ";
    }
```

```
    for (let i = 0; i < tree[level].length; i++) {
      let valStr = formatValue(tree[level][i].value);
      levelStr = levelStr + valStr;
      if (i != tree[level].length - 1) {
        for (let i = 0; i < (2 ** (height + 1)) - 3; i++) {
          levelStr = levelStr + " ";
        }
      }

    }
    console.log(levelStr);
    height--;
  }
}

function prettyPrintV2(root) {
  console.log("Step 3 -- prettyPrintV2():");
  console.log("\n");
  let tree = getAllLevels(root);
  let initHeight = getMaxHeight(root);
  let height = initHeight;
  for (let level in tree) {
    let levelStr = "";
    let lineStr = "";
    let endPair = false;

    let leadSpaces = (2 ** height) - 2;
    for (let i = 0; i < leadSpaces; i++) {
      levelStr += " ";
      lineStr += " ";
    }

    for (let i = 0; i < tree[level].length; i++) {
      let node = tree[level][i];
      let exists = true;
      let nextExists = false;
      let valStr = formatValue(node.value);

      levelStr += valStr;
      if (valStr == "   ") { exists = false; }
      if (i < tree[level].length - 1) {
        let next = tree[level][i + 1];
        if (next.value != '') {
          nextExists = true;
        }
      }
      if (!endPair && exists) {
        lineStr += " ┌─";
      } else if (exists) {
        lineStr += "─┐ ";
      } else {
        lineStr += "   ";
      }

      if (i !== tree[level].length - 1) {
        let betweenCount = (2 ** (height + 1)) - 3;
        for (let j = 0; j < betweenCount; j++) {
          levelStr += " ";
          if (!endPair) {
            if (j === Math.floor(betweenCount / 2) && (exists || nextExists)) {
              lineStr += "┴";
```

```
            } else if (exists && j < Math.floor(betweenCount / 2)) {
              lineStr += "—";
            } else if (nextExists && j > Math.floor(betweenCount / 2)) {
              lineStr += "—";
            } else {
              lineStr += " ";
            }
          } else {
            lineStr += " ";
          }
        }
      }

      endPair = !endPair;
    }
    if (height !== initHeight) {
      console.log(lineStr);
    }
    console.log(levelStr);
    height--;
  }
}

const tree = new BinaryTree();
for (let i = 0; i < 10; i++) {
  const value = Math.floor(Math.random() * 40);
  tree.insert(value);
}


console.log("\n");
prettyPrint(tree.root);
console.log("\n");
prettyPrintV2(tree.root);
```

Console Output:

```
Step 2 -- prettyPrint():

               28
       17              35
   6       26      33      39
 0       18  26


Step 3 -- prettyPrintV2():

               28
       ┌───────┴───────┐
       17              35
   ┌───┴───┐       ┌───┴───┐
   6       26      33      39
 ┌─┘       ┌─┴─┐
 0       18  26
```

A second test:

```
Step 2 -- prettyPrint():
                          22
              7                              34
       5              11              29
                  10              24      32
                            22

Step 3 -- prettyPrintV2():
                          22
              ┌─────────────────────────┐
              7                         34
       ┌──────┐                  ┌──────┐
       5              11              29
                  ┌──┐            ┌──┐
                  10              24      32
                            ┌──┐
                            22
```

Notes:
Our main print function—prettyPrintV2()—first calls getAllLevels() to build the tree into a format that is easily printable. getAllLevels() calls convertToCompleteBinaryTree() (which calls fillTree() and getMaxHeight()), and getMaxHeight(). getMaxHeight() traverses the whole tree recursively; if the tree is very skewed, this operation can take up to O(N). fillTree() visits each node once, filling missing nodes with empty strings, resulting in a complete tree. This is an O(N) operation. So far, we have done two O(N) operations, with N being the number of nodes in the original tree. In the following steps, we will consider N to be the number of nodes in the newly completed tree, consisting of the original nodes and the blank strings that have filled the missing nodes.

Our getAllLevels() function then calls getMaxHeight(), which runs in O(log N) because the tree is now complete. A loop then calls getLevel() twice per level in the tree, or 2 * log N times; getLevel is an O(N) function, so these together are O(N log N).

Back in prettyPrintV2(), we call getMaxHeight() one more time and use a few loops to iterate through each node in the tree and build our strings to print. None of these operations increase the overall Big O time complexity.

All this together simplifies to a total time complexity of O(N log N), with N being the number of nodes (including empty strings) in the tree after it is completed.

LeetCode #208: Implement Trie (Prefix Tree)

A trie (pronounced as "try") or prefix tree is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

Trie() Initializes the trie object.

void insert(String word) Inserts the string word into the trie.

boolean search(String word) Returns true if the string word is in the trie (i.e., was inserted before), and false otherwise. boolean startsWith(String prefix) Returns true if there is a previously inserted string word that has the prefix prefix, and false otherwise.

My solution:

```
class TrieNode {

    constructor() {
        this.children = {};
    }
}

var Trie = function() {
    this.root = new TrieNode();
};

/**
 * @param {string} word
 * @return {void}
 */
Trie.prototype.insert = function(word) {
    let currentNode = this.root;

    for (const char of word) {
        if (currentNode.children[char]) {
            currentNode = currentNode.children[char];
        } else {
            const newNode = new TrieNode();

            currentNode.children[char] = newNode;
            currentNode = newNode;
        }
    }
    currentNode.children["*"] = true;
};

/**
 * @param {string} word
 * @return {boolean}
 */

Trie.prototype.find = function(word) {
    let currentNode = this.root;

    for (const char of word) {
        if (currentNode.children[char]) {
            currentNode = currentNode.children[char];
        } else {
            return false;
        }
    }
```

```
    return currentNode;
}

Trie.prototype.search = function(word) {
    let result = this.find(word);

    if (result && result.children["*"]) return true;
    return false;
};

/**
 * @param {string} prefix
 * @return {boolean}
 */
Trie.prototype.startsWith = function(prefix) {
    let result = this.find(prefix);
    if (!result) {
        return false
    } else return result;
};

/**
 * Your Trie object will be instantiated and called as such:
 * var obj = new Trie()
 * obj.insert(word)
 * var param_2 = obj.search(word)
 * var param_3 = obj.startsWith(prefix)
 */
```

Graph Implementation with Breadth-First Search (Not LeetCode)

```
class Vertex {
    constructor(value) {
      this.value = value;
      this.adjacentVertices = [];
    }

    addAdjacentVertex(vertex) {
      this.adjacentVertices.push(vertex);
    }

    static bfsTraverse(startingVertex) {
      const queue = [];
      const visitedVertices = {};

      visitedVertices[startingVertex.value] = true;
      queue.push(startingVertex);
```

```
    while (queue.length > 0) {
      const currentVertex = queue.shift();
      console.log(currentVertex.value);

      for (const adjacentVertex of currentVertex.adjacentVertices) {
        if (!visitedVertices[adjacentVertex.value]) {
          visitedVertices[adjacentVertex.value] = true;
          queue.push(adjacentVertex);
        }
      }
    }
  }

  static bfsSearch(startingVertex, targetValue) {
      if (startingVertex.value == targetValue) {
          return startingVertex;
      }
      const queue = [];
      const visitedVertices = {};

      visitedVertices[startingVertex.value] = true;
      queue.push(startingVertex);

      while (queue.length > 0) {
        const currentVertex = queue.shift();
        if (currentVertex.value == targetValue) {
          console.log(`${currentVertex.value} == ${targetValue}!`);
          return currentVertex;
        } else {
          console.log(`${currentVertex.value} =/= ${targetValue}`);
        }



        for (const adjacentVertex of currentVertex.adjacentVertices) {
          if (!visitedVertices[adjacentVertex.value]) {
            visitedVertices[adjacentVertex.value] = true;
            queue.push(adjacentVertex);
          }
        }
      }
      return null;
    }
}
```

LeetCode #14: Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings.
If there is no common prefix, return an empty string "".

My solution (JavaScript):

```
/**
 * @param {string[]} strs
 * @return {string}
 */
var longestCommonPrefix = function(strs) {
    let firstWord = strs[0];
    let currentLetter = 0;
    let match = true;
    let prefix = "";
    while ((match == true) && (currentLetter < firstWord.length)) {
        for (let currentWord = 1; currentWord < strs.length; currentWord++) {
            if (currentLetter < strs[currentWord].length) {
                if (strs[currentWord][currentLetter] == firstWord[currentLetter]) {
                } else {
                    match = false;
                    break;
                }
            } else {
                match = false;
                break;
            }
        }
        if (match == true) {
            prefix = prefix + firstWord[currentLetter];
        }
        currentLetter++;
    }
    return prefix;
};
```

Notes:

While my immediate thought was that I could keep track of matching characters as I iterated through each word letter-by-letter, I then thought that the code could run faster if I checked the first letter of each word, the second letter of each word, and so on, terminating immediately as soon as a mismatch was found. I stored the first word in a variable, then used a for loop inside a while loop to test each word's letter at position currentLetter against firstWord's corresponding letter. I believe this is the quickest way I could identify a mismatch.

This algorithm's complexity is essentially O(N*M). It is dependent on two

independent variables: length of the strs, and length of strs[0]. Thus, it does not fall clearly into the linear or quadratic complexity categories.
the best classification is:

- O(NM) → Linear in terms of both N (number of words) and M (prefix length).
- If M is considered a constant upper bound (the LeetCode description specifies 0 <= strs[i].length <= 200), then it simplifies to O(N).

This code's runtime beat 100% of JavaScript submissions on LeetCode.

LeetCode 241: Different Ways to Add Parentheses
Given a string expression of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. You may return the answer in any order.
The test cases are generated such that the output values fit in a 32-bit integer and the number of different results does not exceed 104.

My solution (JavaScript):

```
/**
 * @param {string} expression
 * @return {number[]}
 */
const apply = function (a, b, op) {
  if (op === "+") {
    return Number(a) + Number(b);
  } else if (op === "-") {
    return Number(a) - Number(b);
  } else {
    return Number(a) * Number(b);
  }
};

var diffWaysToCompute = function (expression) {
  const operators = new Set(["+", "-", "*"]);
  let results = new Array();
  let base = true;
  for (let i = 0; i < expression.length; i++) {
    if (operators.has(expression[i])) {
      const left = diffWaysToCompute(expression.slice(0, i));
      const right = diffWaysToCompute(expression.slice(i + 1, expression.length));
      left.forEach(leftRes => {
        right.forEach(rightRes => {
            let result = apply(leftRes, rightRes, expression[i]);
            results.push(result);
        })
      })
      base = false;
```

```
    }
  }
  if (base) {
    return [Number(expression)];
  } else {
    return results;
  }
};
```

Notes:
I employed recursion to traverse all of the potential routes for calculation, and wrote a helper function (apply()) to parse operator symbols and make calculations.

LeetCode #88: Merge Sorted Array
You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.
Merge nums1 and nums2 into a single array sorted in non-decreasing order.
The final sorted array should not be returned by the function, but instead be stored inside the array nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

My solution (C#):

```
public class Solution {

    public void Merge(int[] nums1, int m, int[] nums2, int n) {
        bool sorted = false;
        for (int i = m; i < (m + n); i++) {
            nums1[i] = nums2[i - m];
        }
        while (!sorted) {
            sorted = true;
            for (int i = 0; i < (m + n - 1); i++) {
                if (nums1[i] > nums1[i + 1]) {
                    int temp = nums1[i];
                    nums1[i] = nums1[i + 1];
                    nums1[i + 1] = temp;
                    sorted = false;
                }
            }
        }
    }
}
```

Notes:
There are two main tasks here: merge the arrays and sort them. The simplest way, the way that came immediately to mind, is to simply replace the zeroes in nums1[] with the values in nums2[] and then sort them afterwards. I simply replaced each nums1[] value after place m with the values in nums2[], then did a bubble sort to order the values from least to greatest. The bubble sort in C# required a temporary variable.

Due to the use of a bubble sort, this algorithm's Big O time complexity category is O(N^2).

With O(N^2) time complexity, my solution ran faster than 32.24% of submitted solutions. Not very impressive.
Here is an example of a better, faster solution:

```
public class Solution {
    public void Merge(int[] nums1, int m, int[] nums2, int n) {
        int l = m - 1 ;
        int r = n - 1;
        int k = m + n - 1;

        while(r >= 0 && l >= 0)
        {
            if(nums2[r] > nums1[l] )
            {
                nums1[k] = nums2[r];
                r--;
            }
            else
            {
                nums1[k] = nums1[l];
                l--;
            }
            k--;
        }

        while(r >= 0){
            nums1[k] = nums2[r];
            r--;
            k--;
        }
    }
}
```

This algorithm uses variables to point towards relevant positions in the arrays. k iterates through nums1[], starting at the very end, and compares the last unsorted values from the pre-filled values of nums1[] and nums2[], using l and r to iterated through the pre-filled values of each array, starting at the end. Because these arrays are assumed to be pre-sorted, the only sorting the algorithm must do is decide which array's value to place at the k position in nums1[].

The time complexity of this algorithm is O(N), which is far preferrable to O(N^2). The better algorithm cleverly uses variables to avoid nested loops and takes advantage of the fact that each array is pre-sorted to reduce steps.

LeetCode #13: Roman to Integer

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

Symbol	*Value*
I	*1*
V	*5*
X	*10*
L	*50*
C	*100*
D	*500*
M	*1000*

For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

- *I can be placed before V (5) and X (10) to make 4 and 9.*
- *X can be placed before L (50) and C (100) to make 40 and 90.*
- *C can be placed before D (500) and M (1000) to make 400 and 900.*

Given a roman numeral, convert it to an integer.

My solution (JavaScript):

```
/**
 * @param {string} s
 * @return {number}
 */
var romanToInt = function(s) {
    const romanMap = new Map();
    romanMap.set("I", 1);
    romanMap.set("V", 5);
    romanMap.set("X", 10);
    romanMap.set("L", 50);
    romanMap.set("C", 100);
    romanMap.set("D", 500);
    romanMap.set("M", 1000);
    let value = 0;

    for (let i = 0; i < s.length; i++) {
        if ((romanMap.get(s[i]) < romanMap.get(s[i+1])) && ((i+1) < s.length)) {
            value -= romanMap.get(s[i]);
```

```
        } else {
            value = value + (romanMap.get(s[i]));
        }
    }

    return value;
};
```

Notes:
My algorithm processes each character of the string once. Since each part of the process happens in constant time, the overall time complexity is O(n), where n is the length of the input string.

The space complexity is O(1) because the Map always stores a constant number (7) of key-value pairs, regardless of the input size.

LeetCode #9: Palindrome Problem
Given an integer x, return true if x is a , and false otherwise.

My solution (JavaScript):

```
/**
 * @param {number} x
 * @return {boolean}
 */
var isPalindrome = function(x) {
    let xstring = x.toString();
    let palindrome = true;
    for (let i = 0; i < xstring.length/2; i++){
        if (xstring[i] != xstring[xstring.length - (i + 1)]) {
            palindrome = false;
        }
    }

    return palindrome;
};
```

Notes:
Time complexity is O(n): converting the number to a string takes O(n) time, where n is the number of digits. The for-loop then iterates over the string once, resulting in overall linear time.

Space complexity is O(n): the string representation of the number is stored in memory, which requires O(n) space.

LeetCode #169: Majority Element
Given an array nums of size n, return the majority element.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

My solution (JavaScript):

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var majorityElement = function(nums) {
    let counts = [[nums[0], 1]]
    let majority = nums[0];
    let foundMajority = false;
    if (nums.length > 1) {
        for (let i = 1; i < nums.length; i++) {
            let exists = false;
            for (let j = 0; j < counts.length; j++) {
                if (counts[j][0] == nums[i]) {
                    counts[j][1]++;
                    exists = true;
                    if (counts[j][1] > (nums.length / 2)) {
                        majority = counts[j][0];
                        foundMajority = true;
                    }
                    break;
                }
            }
            if (exists == false) {
                counts.push([nums[i], 1]);
            }
            if (foundMajority == true) {
                break;
            }
    }
    }
    return majority;
};
```

Notes:

In the worst-case scenario, where almost every element is distinct, the outer loop runs O(n) times and for each iteration the inner loop may iterate over up to O(n) elements in counts. This yields an overall worst-case complexity of $O(n^2)$.

In the worst case the counts array could store an entry for each unique element in the input array. Therefore, the space complexity is O(n).