Merge Intervals
(https://leetcode.com/problems/merge-intervals/)

Code:
```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort()
        newIntervals = []
        maxVals = []
        for i in range(len(intervals)):
          maxVals.append(intervals[i][1])
        end = max(maxVals)
        i = 0
        finalStart = intervals[0][0]
        while True:
          if intervals[i][1] == end:
            break
          if intervals[i][0]<intervals[i+1][0] and intervals[i][1]>intervals[i+1][1]:
            del intervals[i+1]
          else:
            if intervals[i][1]<intervals[i+1][0]:
              newIntervals.append([start,intervals[i][1]])
              finalStart = intervals[i+1][0]
            i += 1
        newIntervals.append([finalStart,end])
        return newIntervals
```

Runtime: 88 ms

Memory Usage: 16.2 MB

Explanation: I first sorted the intervals given numerically and iterated through the 2D list so that I could add all the end values of the intervals to a list. With the list, I can find the end value so I can have the largest number among all the intervals, and the start value of the sorted intervals list is the smallest number among all intervals to start with. In a while loop, I first made sure to check if the end value of the current second index is the max value to know when to break the loop which also covers the edge case where the entire list is just 1 interval. Then, I made sure to check if the interval afterward is inside the current interval (e.g. if the current interval is [1,4] and the next one is [2,3]) so that I could delete the inside interval since it automatically overlaps. If that's not the case, then I check if the end of the current interval is less than the start of the next interval since that means that the current interval and the next interval don't overlap. If they don't overlap, then add the current interval to a list of new non-overlapping intervals. Also in that block, as it's iterating through the loop, it's constantly updating the finalStart variable since the program needs to know what's the first index of the final interval when the loop ends. The program can't do that part on its own since at the end, there's no interval after the last interval to compare it to and ultimately the program returns the new list without any overlapping intervals.

Longest Substring Without Repeating Characters
(https://leetcode.com/problems/longest-substring-without-repeating-characters/)

Code:
```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        j = 0
        subLength = 0
        letterAppearances = {}
        for i in range(len(s)):
            j = i
            while True:
                if len(letterAppearances) >= j-i and j < len(s):
                    letterAppearances[s[j]] = 1
                    j += 1
                else:
                    if len(letterAppearances) > subLength:
                        subLength = len(letterAppearances)
                    letterAppearances = {}
                    break
        return subLength
```

Runtime: 498 ms

Memory: 16.4 MB

Explanation: This question required me to use a dictionary to record how many times each letter appeared in the string as I iterated through it. So using a nested loop where j starts where i is, I checked each possible unique substring by adding a character to the dictionary if the length of the dictionary was greater than or equal to the length of the substring because dictionaries don't allow duplicates. If it didn't meet that condition or it iterated to the end of the string, I'd replace the current counter of the length of the largest substring with whatever the length of the current dictionary is if it's larger, clear the dictionary, and break the inner loop. After reaching the end of the string, the program returns the length of the largest substring.

Sum Root to Leaf Numbers
(https://leetcode.com/problems/sum-root-to-leaf-numbers/)

Code:
```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def sumNumbers(self, root: Optional[TreeNode]) -> int:
        stack = [[root, 0]]
        totalSum = 0
        while stack:
            node, pathSum = stack.pop()
            if node:
                stack.append([node.right, pathSum*10+node.val])
                stack.append([node.left, pathSum*10+node.val])
                if stack and node.left == None and node.right == None:
                    totalSum += stack[-1][1]
        return totalSum
```

Runtime: 78 ms

Memory Usage: 14.1 MB

Explanation: Since this question requires depth-first search, I decided to use a stack to traverse the binary tree. The stack contains arrays of the node and the sum of the root-to-leaf path for that node. I immediately started the stack with the root of the binary tree along with the starting value of 0 and made it so that while there are still arrays in the stack, pop the last array from the stack. Using this popped array, I added both the right and left node of it to the end of the stack and repeated this process until the program comes across a node with no children. While this process is going on, I'm also updating the sum of the path by adding the value of the node to the end of its parent's node which eventually gets added to the total sum of all the paths to be returned at the end of the program.

Remove Nth Node From End of List
(https://leetcode.com/problems/remove-nth-node-from-end-of-list/)

Code:
```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
        head2 = head
        counter = 1
        while head2:
            counter += 1
            head2 = head2.next
        dummy = ListNode()
        cur = dummy
        while head:
            counter -= 1
            val = head.val
            if head:
                if n != counter:
                    cur.next = ListNode(val)
                    cur = cur.next
                head = head.next
        return dummy.next
```

Runtime: 30 ms

Memory Usage: 16.2 MB

Explanation: First I wanted to find the length of the linked list to use as a counter, so to get it I just iterated through a copy of the linked list because I didn't want to change the original yet. Since the program wants the nth node removed from the end of the linked list, I used a while loop where the node keeps going forward but the counter goes backward. With this, I made a dummy linked list which will end up being the final linked list with the removed node and a variable called cur which is the current node of the loop. So until the end of the loop and if head exists, keep adding to the dummy linked list unless it reaches n, in which case skip it. Finally, the programs return dummy.next to return its head.

Nearest Exit from Entrance in Maze
(https://leetcode.com/problems/nearest-exit-from-entrance-in-maze/)

Code:
```
class Solution:
    def nearestExit(self, maze: List[List[str]], entrance: List[int]) -> int:
```

```
    queue = []
    exitFound = False
    def traverse(x,y,steps):
      steps = maze[x][y]
      if x == 0 or y == 0 or y == len(maze[0])-1 or x == len(maze)-1:
        if steps > 0:
          return steps
      if x > 0:
        if maze[x-1][y] == ".":
            queue.append([x-1, y])
            maze[x-1][y] = steps + 1
      if y > 0:
        if maze[x][y-1] == ".":
            queue.append([x, y-1])
            maze[x][y-1] = steps + 1
      if x < len(maze)-1:
        if maze[x+1][y] == ".":
            queue.append([x+1, y])
            maze[x+1][y] = steps + 1
      if y < len(maze[0])-1:
        if maze[x][y+1] == ".":
            queue.append([x, y+1])
            maze[x][y+1] = steps + 1
      return 0
    maze[entrance[0]][entrance[1]] = 0
    traverse(entrance[0], entrance[1], 0)
    while queue:
      curr = [queue[0][0], queue[0][1], 0]
      queue.pop(0)
      exitFound = traverse(curr[0], curr[1], curr[2])
      if exitFound:
        break
    if exitFound == False:
      return -1
    return exitFound
```

Runtime: 748 ms

Memory Usage: 17.2 MB

Explanation: This problem requires dynamic programming to traverse through a maze in the form of breadth-first search, so the most efficient way to handle it is by using a queue. I made a recursive function that takes in the x and y coordinates of a square in the maze, as well as the steps taken away from the starting point. By default, the starting position instead of '.', I replaced it with the initial value of 0 for its steps and I also put that position along with the steps into the queue as an array. Firstly, the function checks if the starting position is an exit and makes sure to search for a different exit properly if it is (because there are conditions for when the starting position is at a wall to not proceed if it can't look at a position outside the matrix). To look for an exit, the function looks at the squares around it to see if the square is a '.' and if it is, then replace that spot with a counter of the steps from the start so that it can be treated as a wall (so infinite backtracking can't happen) and add new the spot/steps to the queue. Eventually, if an exit is found then return the steps needed to reach the exit, otherwise return false. The queue removes the last array put into the list, runs that array through the function, and keeps repeating these steps as long as there are arrays in the list or

if one of the function calls returned a non-false value. Then if no exits are found by the
time the queue is empty, return -1 and otherwise return the steps when a wall was reached.