3. Longest Substring Without Repeating Characters

In this problem, we have to simply output the longest *length* of a **contiguous** substring that has no duplicate characters. The best data structure to achieve this is a set, which never allows duplicates. So, we write a sliding window algorithm as follows:

- First, we loop through the given string
- As we loop through, the set will serve as our "window" which we will shift as we traverse the string
- Then, we have two pointers on the bounds of our window:
    - A left one, which will track the left side of the substring
    - A right one, which loops with the substring's right side
- As the loop processes new characters in the string through the right pointer, there may be elements already in the set.
    - Therefore, we will need to keep removing elements from the left end (for contiguous *sub*string) until the new element is no longer in the substring set, in order to avoid duplicates.
    - This is because the duplicated character may not always be on the left, but could be in the middle. So if we have substring "abcd" and the next character is "c", then three characters need to be deleted, leaving us with "d".
    - At each removal, the left pointer increments. From the example, this brings it to the index at 'b', 'c', and then 'd'.
    - After all removals, the new character (which was previously duplicated) can now be added. From our example, this would be 'c', leaving us with "dc."
    - At the end of each loop iteration (1 new character element processed), we will need to keep track of the length of the duplicate-less substring that we have. We have this through the difference between the right and left pointer.
    - Run a max algorithm (i.e. update the final length with a max function at each loop iteration) to get the longest possible non-repeating substring.
- **Analysis:** the resulting time complexity is O(n), and space complexity is also O(n)

---

4. Median of Two Sorted Arrays

This problem may be called "hard," but let's be *honest* here, it is **trivial**. The median of two sorted arrays is the median of the merged array. Apply the classic "merge" function from merge sort in a new array and bam, you can get a median. Instead of the usual "divide and conquer" you are just "conquering," which further trivializes the problem. How is this hard?

Forgot how to do the merge function? Here it is:

1. Create a new array/vector for the merged arrray
2. Initialize two pointers: i the first array, and j in the second, with both starting at zero. Both i and j are *outside* any loops.
3. Run the loop until either i or j has reached respective array limits. In each iteration, check which of the first or second arrays at respective i/j index is larger and enter the smaller element into the merged array.

4. After the loop is done, we copy in the remaining elements of both arrays into the merged array, each with its own loop.
    - The way this works, is that if the size of one is larger than the other, only one of these loops will run. For the other, the index i or j will have already reached the array size limit.
    - Since we know each array is sorted, the remaining "push" of elements will still be sorted.
5. Finally, we take the median of the resulting array, doing the correct procedure to account for odd and even-sized arrays.

Analysis: the worst case time complexity is O((n+m) * log(m+n)). The space complexity is obviously O(n+m), where n and m are the sizes of the arrays.

---

7. Reverse Integer

In this problem, we take an integer, and reverse its digits, while retaining the sign.

First, we will apply the classic algorithm known already:

1. Create a new integer (the reversed one) and assign it to zero.
2. Loop through the given integer, each time truncating the final digit by dividing by ten.
3. In each iteration, multiply the current result by ten, and add the input mod 10. This adds the rightmost digit of the input to the right of the result.
4. As the digits tack up, you will get the reverse.

Finally, once we are done, we account for this problem's possible input outside the 32 bit range.

Analysis: the worst possible time complexity should (?) be O(n), where n is the number of digits. The more digits, the more time. The space complexity is O(1).

---

46. Permutations

This is a classic problem for backtracking In any array, we have a tree of possible permutations. This can be obtained by swapping different pairs of elements, through which different permutations appear with different pairs swapped.

We run our recursive backtracking algorithm as follows:

- We systematically swap by going through the first pair, and doing the next, and so on, traversing to the bottom of permutation "tree".
- However, at the final swap, we will have to backtrack in order to process permutations with other swaps, so we have to swap it back.
- To write the code, there will be a recursive function that can swap, recurse, and swap back.
- At the bottom of the tree, we have finished this round of swapping and found a permutation

---

94. Binary Tree Inorder Traversal

For this problem, we are required to produce a binary tree's nodes inorder traversal. That is, we must go from the bottom of the left subtree to the root, and then end up at the bottom of the right subtree. Each run is traversed left-root-right, if we decide to view each subtree like a parent with two children.

How do we traverse like this? Of course, we will discuss the iterative solution, as the recursive solution is seriously *easy*. You know, you just recur the left, print the current, then recurr the right. This works for any subtree of a given tree. This is trivial and is based on its definition.

To solve it **iteratively**, we must do the following: First, the roots must be stored in a stack of nodes. Then, we do the following, starting from the root of the main tree itself as the current node:

1. Push the current node in the stack, and enter the left child node.
2. Keep repeating step #1 until the left node is null.
3. Then, the stack should probably not be empty for a normal tree. So we take the top item in the stack, enter its right node, and pop that node.
4. Then, we keep repeating steps 2 and 3 until the stack is empty and we are done.

Analysis: the time complexity for this algorithm is O(n), where n is the number of nodes in the tree. The space complexity is also O(n), where n is the height of the tree.

Disclaimer: I just guessed the time and space complexities in the analysis

---

196. Delete Duplicate Emails

This databae problem can be solved with SQL by doing: (thanks ChatGPT for the analysis)

1. A `DELETE` statement indicating that you would like to delete entries from table `Person`. You should give this instance of the table an alias.
2. An `INNER JOIN` statement between your first alias in the statement from number one and another instance of the same table with a different alias. This basically compares the table's entries with itself. Instead of selecting an inner joined result, overall, this makes you delete the actual entries.
3. Then, you will set the conditions for the deletions. In this case, you want to delete the entry in your first instance from step #1 that has a larger id number than an entry in the alias from step #2. Another condition is that they must be duplicates, i.e. the emails are the same.
4. Don't forget a semicolon!

The code: (I'm only doing this for DB problems, hopefully) `DELETE p1 FROM Person p1 INNER JOIN Person p2 WHERE p1.id > p2.id AND p1.email = p2.email;

---

300. Longest Increasing Subsequence

In this Longest Increasing Subsequence (LIS) Problem, I use an optimized approach to reduce the space complexity. I follow the steps as below:

1.
