1. Two Sum

For this two sum problem, the naive/brute force solution is to make a nested loop through the array, and check if there exists a pairt that adds to the target, which is in the lowest 5%. However, there are many much more optimized solutions. Let's consider one of them, which can use a hash table (unordered_map):

1. Iterate through the array, from left to right.
2. For each element nums[i], calculate the complement by subtrating it from the target, with formula complement = target - nums[i].
3. If complement exists in hash table, we have found a solution, which should be collected.
4. If it does not, add the current iterated element of the array, nums[i], as a key in the hash table, with the index as its value.
5. Repeat steps 2-4, until we find a solution. Why does it work? Since every non-summable element at the start will (hopefully) find its complementary sum value to the target in the hash table later on, so we should store it.
6. If no solution is found, return an empty array.

Analysis: this optimized solution has a time complexity of O(n), and the worst possible space complexity is O(n), where n is the number of elements in the given array.

---

2. Add Two Numbers

In this problem, we have to take two numbers, whose digits are stored in reverse, in linked lists, and return a linked list for the sum of the numbers, that is also in reverse order. To do this, we must create a new linked list, that is to be returned. The node to follow is the tail, which starts out at its head.
Then, we iterate through both linked lists at the same time, as follows:

1. If neither linked lists have elements at the current iteration/index, and there is no carry, we can stop running. Start with a carry of 0.
2. For each iteration/index, we take the value contained by each number. However, we need to check whether they are null, then assign 0 or the corresponding number at that index.
3. Then, we sum the current carry, and values from corresponding indices of each number.
4. Assign a new carry, and the value of the sum at that index.
5. At each iteration, we create a new node (i.e. in C++, dynamically allocate), and assign the digit value to that node's value.
6. Finally, we return the the next of the original null head, which corresponds to the lowest digit of the summmed integer.

Analysis: the time complexity will be O(max(m, n)) where m, n are the sizes of the numbers. The space complexity is O(1), as the length of the returned list is always max(m, n) + 1.

---

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.

5. Longest Palindromic Substring

In this DP problem, we have to find the actual palindromic substring of the given string that is of the longest possible length. By a simple of analysis of a two pointers (ends of palindrome) brute-force algorithm, we find it takes O(n^3) of time complexity, which is not possible.

Therefore, we must use memoization. We do this by seeing that in a string, if the substring between indices i, j, then so are the substrings for i+1, j-1, and so on. The converse is also true. If the characters at i and j are the same, and substring i+1, j-1 is a palindrome, then i, j must also be. Another important note is that the lowst possible length of a palindrome is 1, and even numbered sizes start where substring i, i+1 is a palindrome.

We run a DP algorithm as follows:

1. To store palindrome/non palindrome states for different upper/lower bounds, the obvious choice is a 2D array of booleans.
2. We populate the dp array based on the principles discussed earlier.
    - The shortest palindrome is of length 1, so the main diagonal at indices (i, i) must be set to true.
    - Iterating through the string, if two adjacent characters are equal, that is a palindrome, and attributes of the substring there should be collected accordingly.
3. Then, we only need check for palindromes of length 3 and above. By setting upper/lower bounds and using the rules for expansion of the palindrome's length from earlier, we can update the substring at each iteration.
4. Return the resulting string.

Analysis: The above algorithm's time complexity is O(n^2), which is still not great, but we can make it better. Since we populate the entire dp table, setting each one takes O(1) time. Its space complexity is also O(n^2) given the dp table's size. **Exercise:** learn Manacher's algorithm, to solve the problem in linear time.

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).

11. Container With Most Water

This problem's solution will involve the use of two pointers. The algorithm should run as follows, with the use of a left and right pointer. We will also have a holding variable for the maximum area.

1. At the current left and right pointers, compute the area by multiplying the index distance by the minimum height of the "walls." Update the maximum area accordingly, if the current area is larger.
2. Then, if the left wall is shorter than the right wall, the we increment the left wall, in the hopes that we may find a higher one. .
3. However, if the right wall is shorter than the left wall, we decrement the right, hoping to get a higher one as well.
4. Steps 1 - 3 continue until the left and right pointers converge, when the algorithm should terminate.

Analysis: the time complexity of the algorithm will be O(n), and the space complexity should be O(1). This is because the bare minimum of memory needed, is the two pointers and the maximum area.

20. Valid Parentheses

This easy problem is a great one to practice using a stack. ...TBC

22. Generate Parentheses

This is a backtracking problem...TBC

42. Trapping Rain Water

In this extension of the previous problem (https://leetcode.com/problems/container-with-most-water/description/), we must find the total area encompassed by a given set of fixed walls, which entails a variation of a two-pointers solution. The steps are as follows:

1. Maintain 4 pointers. Two for the left, and two for the right.
2. ...TBC

46. Permutations

This is a classic problem for backtracking to generate permutations of a an array of numbers. 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:

- By using the helper function (a recursive one), we pass in the array, along with a left pointer and right pointer placed at the start and end of the array, respectively.
    - However, we will be primarily focused on shifting the left one.
    - Eventually, the recursive call of the funciton should make a permutation tree.
- Then, we iterate from the left pointer to the right pointer.
    - In each iteration, we first swap the values of the array at the left with the pointer iterating between the left and right.
    - Then we recursively call the function, but with the argument for the left pointer one more to the right.
    - After the recursive call, we swap it back to **backtrack** before the next iteration.
    - When the left pointer has moved to the same spot as the right pointer, there is nothing more to swap, so at the leaf of the tree, the function should collect result and end the function call.
    - To give an example of the letters "abc", the first recursive call swaps 'a' with 'a', so we then have a left pointer at 'b', so we must swap 'b' and 'c', leaving us with 'acb', but now the left and right pointers are at the same spot. Therefore, we are at the leaf of the tree and must begin backtracking.

Analysis: the time complexity of the backtracking will be O(N*N!), since there are N! permutations for an array of size N, and it takes N steps to get to each one. **Image of Tree** : If you count the number of steps taken with the backtracking, from root to all the leaves and back to the root, it is a total of 18 steps!

69. Sqrt(x)

In this classic binary search problem, we run through the following steps with the given integer. Since power and exponents are prohibited, we must use binary search and pure multiplication.

1. Set a left and right pointer at 1 and `x` respectively.
2. Program a binary search with the two pointers.

- With each `mid` pointer, check if `mid` is also `x/mid`, which would make `mid` the square root.
- If `mid` is less than `x/mid`, then we have two options. Since the problem wants it to be rounded down to the nearest integer, we temporarily set the final result to the mid, then adjust the right pointer to `mid-1`.
- Otherwise, it is out of range, and the left pointer must be incremented.
- *Analysis:** the above algorithm takes a time complexity of O(log(n)), the standard time complexity of a binary search. Its space complexity is O(1), since we use constant memory.

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

98. Validate Binary Search Tree

This tree traversal problem...

104. Maximum Depth of Binary Tree

This is an obvious DFS problem, as we want the maximum depth of a binary tree, so we must search by depth. So we will simply do the following, while keeping the current depth as a parameter in a recursive DFS function call:

1. Check if the current node is null/empty. If it is, return zero, which helps handle a corner case.
2. Check if the left node is not null, and if that is satisfied, recursively call the DFS function with the left node, and an incremented depth.
3. Do the same as mentioned in step 2 for the right node. However, we must have a counter to take the maximum of the results from the left and right nodes.
4. Otherwise, if we are at the leaf and both left and right nodes do not exist, then we return the result we currently have, whether 1, 2, 3, etc....
5. Finally, we return the maximum of the left and right subtree depths taken from steps 2 and 3.

Analysis: this algorithm's time complexity is O(n), the standard for DFS through a binary tree. The space complexity is O(h), where h is the height of the tree, since the memory used at any one moment will take up at most the maximum depth of the tree.

---

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;

---

226. Invert Binary Tree

In this easy tree problem, we must horizontally invert a binary tree, given its root node. We will accomplish this via a BFS traversal with a queue. Why do we do this? Because we want to invert level-by level, and switch the left and right pointers at each level. Steps are:

1. Push the root node to the queue, loop until queue is empty.
2. In each iteration, pop the front, and swap left/right pointers.
3. Then, if left/right pointer(s) are not null, push them into the queue.
4. At the end, you will still have root point to root, you didn't move it to the left/right, so return the root pointer.

Analysis: time complexity of the algorithm is O(n), where n is the number of nodes in the tree. The space complexity is also O(n), ***but*** where n is the maximum *width* of the tree. The more width, the more nodes stored at each level.

257. Binary Tree Paths

In this problem, we are required to print all root-to-leaf paths, in the form of a string "a->b->...->c", where a is the root, c is the leaf, and all nodes in between are the path taken. We will run a DFS algorithm. It will take in the root node pointer and the string representing the path, among other things. The algorithm should work as follows:

1. Check if left child is not empty, and if true, then recursively run the DFS, adding a "->" and the value of the node to the path string, and moving the root node in the recursive call to the left node.
2. Do the steps in #1 to the right node.
3. If both the left and right child nodes are empty, you are at a leaf, and cannot add anything else. So we must add the path string to the list (which must be updated in all recursive calls).
4. When done, return the list of path strings.

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

300. Longest Increasing Subsequence

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

In the general LIS problem, the goal is to take the a set of elements from array, i.e. from any index but in original order that increases from left to right.

The standard DP algorithm would run as follows:

1. Create an array with corresponding indices to store the longest increasing subsequence up to that number.
2. Populate the dp array with all 1, as that is the longest increasing subsequence at worst.
3. Then at each index, make another loop, from index 0 to that index,
4. In the nested loop, check if the lower index is smaller than the current index, and if its dp index is also not to small as to decrease the dp element at the current index.
5. Then, if the conditions are satisfied, the two elements are "linked" and the increasing subsequence for that pair of indices is extended.
6. At the end, there may be longer increasing subsequences in the middle than at the end, so we have to return the max value in the dp array.

Analysis: the time complexity is O(n^2), since there is a nested loop. The space complexity is O(n), as we create a 1D array with exact same size as given array.

However, we can make an even more **optimized algorithm**, that utilizes binary search:

1. We have an array, which we call `lis` here to store our subsequence, which is first initialized with the first element of the `nums` array.
2. Then, as we loop through the `nums`, we check two things for each element.
    - If it is larger than the last element of `lis`, then it is an increasing subsequence and is appended to `lis`.
    - However, if it is smaller, then two things could happen. Either it is just smaller than the last element, but bigger than the second last, or its order is way up front and has no significance to the subsequence since it's out of order.
    - The general way to approach it, is by finding the element of `lis` at the smallest possible index that isn't smaller than the element at `nums`. In C++, this can be achieved with the `lower_bound` function, which returns an iterator.
3. Finally, at the end, the `lis` array should have our longest increasing subsequence, the size of which should be returned.

Analysis: this algorithm's time complexity is O(n*logn), where n is the number of elements. Its worst possible space complexity is O(n), the case where `nums` is already an LIS.