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 subtracting 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.

---

6. Zigzag Conversion

In this string manimpulation problem, we will approach it by splitting the problem into the first row, the middle rows, and the last rows, since these 3 parts have different behaviours. Overall, we simply create a new string, which is to be returned, and concatenate character by character. The steps are as follows:

1. In the first row, take the first element of the string, and skip by (numRows-1)*2 each time, concatenating the corresponding string character at each step.
2. For the middle rows, behaviours and skipping patterns change depending on the "column" used.

- For even numbered times of concatenating a character to a row, you skip by (numRows - i - 1)*2, where i is the row number.
- For odd numbered times, you skip by i*2, where is is also the row number.
- This is because for various rows, you will have to skip different amounts through the given string at different times. See the examples in the problem..

3. For the last row, follow the same procesure as the first, but start at the numRows - 1 element of the string instead.

Analysis: the time complexity of this algorithm is O(n), since each element is accessed once. The space complexity is O(1), as only 1 extra string is used.

---

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 example to practice using a stack. It will involve the traversal of the given string, and the storing of brackets in a stack. The steps in the traversing loop are as follows:

1. If a character is an open bracket of any form, push to the stack.
2. Otherwise, you pop the stack, provided that: a) the stack is not empty b) the top of the stack doesn't correspond with the current bracket (which must be ')', ']', or '}', since step 1 covered all open brackets).
3. Then, at the end of the traversal, if the stack is empty. The string is valid. All open-close bracket pairs have entered and exited the stack accordingly.
4. Otherwise, if there are still elements, it is not valid.

Analysis: this very simple algorithm runs on O(n) time, and uses O(n) memory, where n is the size of the string.

---

22. Generate Parentheses

This is a backtracking problem to generate a set of all possible strings with a given number of pairs of parentheses. We will make use of a `void` helper function called `solve` which will take in `n` twice, as well as a holder string, and an array holding the result to be returned. This allows us to keep track of the number of remaining open/close brackets. In that function, we do:

1. If there are no more brackets to concatenate (open or close), push the current string to the resulting array, and cut out the function call.
2. If there are the same number of open brackets available as close, or if there are no more close brackets available, then append an open bracket and recursively call. Don't forget to *decrement* the number of available open brackets. *Side Note:* the solution will still be correct if you don't consider whether there are no more close brackets or not, it just happens to run slower, but I don't know *why*.
3. If there are no more open brackets available, we have only the choice to append a close bracket and continue the recursive call.
4. Otherwise, everything is normal, and we can pursue two paths by making two recursive calls. The first adds an open bracket, the other appends a close bracket. .

Analysis:: the time complexity is $O(N*2^N)$, where N is 2*n, according to this (https://leetcode.com/problems/generate-parentheses/solutions/3290261/i-bet-you-will-understand-intutive-solution-beginner-friendly-c/). This is because the size of every string in the returned array is 2*n, and there are 2^N strings. The space complexity is O(n).

---

23. Merge k Sorted Lists

In this problem, we apply a divide-and-conquer approach to merge the k given sorted lists, with the following steps:

1. If k is zero, return null. There are no lists to merge.
2. Otherwise, call a helper function `mergeSort`, which finds the middle list which we will call `mid`, and recursively calls `mergeSort` on all lists from the leftmost and `mid`, and from `mid+1` and the rightmost list.
3. Each `mergeSort` recursive call should return a `ListNode*` of its own. After that, another helper function, `merge` should merge the two `ListNode*` linked lists.
4. This recursive call will eventually narrow down to groups of pairs of lists. Which should gradually merge to one list.
5. In function `merge`, we run as follows:

- Create a dummy, null new node, which will hold the merged lists's head. This is because once you run through a linked list, you cannot go back, so a node must be kept at the head to hold it.
- Create another node at the address of the dummy node, which will follow the concatenated elements.
- Follow the classic `merge` from merge sort. One tip for efficiency is: when accounting for unequal list sizes, there is no need to continuously loop. Just tap on the rest of one of the two lists, which already points to the remaining elements not yet merged into the returned ListNode.
- At the end, return the next of the dummy node, as the dummy itself is null. In addition, don't return the next of the traversing node, as that is always going to be `nullptr`
- *Analysis:** the time complexity of this algorithm is ... pretty complex. While the standard merge sort time complexity is O(N*log(N)), this solution considers multiple lists of different sizes, hence the new dependence on K as well. The space complexity of the algorithm is O(N), where N is the total number of elements in all the lists.

---

35. Search Insert Position

In this easy problem exercising binary search, we must find the index of a number in an array, or its index had it been in the array. So while the classic binary search algorithm is like:

1. We start with two pointers, a left pointer at the start, and a right pointer at the end.
2. A middle index, called `mid` holds the middle of the two pointers.
3. Then, looping while left <= right, we check if mid is the target value. It it is, we return the *index* at mid. Otherwise, depending on the position of the value at mid relative to the target value, we adjust the pointers to make the next round of searching only to the left or right half of the array. Remember, the array is *sorted*!

However, for this problem, the main modification is to have a temporary index to hold, as an index must still be returned at the very end even if the target value does not exist. Note first, that the mid value will 'tilt' towards the left, as all integer casting removes decimal points.

To account for this, if the value at the mid pointer is less than the target, we must assign the temporary index *one index to the right of it*. With some basic intution, this is especially helpful if the target is larger than all values in the array. On the other hand, if the value at the mid pointer is larger than target, then the temporary result is at *that same index*. For instance, for an array `[1, 3]`, with target value 2, the mid would be at 1, but then would move to 3. At that point, the 2 would have been at index 1, currently held by 3.

Analysis: while this algorithm is a modification of binary search, the changes are purely cosmetic, and do not change the time complexity of the algorithm, which is of course, `O(log n)`. The space complexity is `O(1)`, as constant memory is allocated at each run.

---

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. The two left pointers are one element apart and so are the right pointers. One of each is pointing at the respective boundaries of the height array, the other is directly adjacent.
2. We then traverse the height array by

---

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!

---

55. Jump Game

This is a classic greedy problem, in which only the most convenient solution is pursued. We will run a solution that traverses the array once, keeping track of the number of steps allowed (with a variable initialized to 1). Here are the steps:

1. Since each new array element introduces a possible new number of steps allowed, we must update the number of steps that we are allowed to jump.
2. If the number of allowed steps at the current index is larger than the current value, we update it. Otherwise, in cases such as stepping over a element 0, we keep going. This variable storing the number of steps allowed is decremented at each iteration, every new step means we have 1 fewer steps allowed to travel.
3. If that number of allowed steps is zero and we are not yet at the end, we cannot reach the end.
4. However, if we are at the end and the previous condition is false, then we can definetly reach the end.
5. Otherwise, we increment, as we are within the current allowed number of steps and we are not yet at the end of the array.

Analysis: the time complexity is O(n). The space complexity is O(1). Only 1 new variable is allocated to track the number of allowed steps remaining, and an index counter.

---

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.

---

78. Subsets

Backtracking solution: TBC

Bitmasking solution: Since there are 2^n possible subsets, we create a C++ bitset of size n (as we need n bits) and iterate through all possible numbers from 1 to 2^n - 1. The zero bitset is the empty subset, and the full bitset (n ones) is the full set. The steps are:

1. As previously mentioned, there are 2^n bitsets needed to represent the subsets (which are mathematically, combinations of the array, where each element can either appear [1] or not appear [0]).
    2. So we iterate from 0 to 2^n-1, each time assigning our single bitset to that number.
2. Then, as each bitset has indices, we iterate through it. If a certain bit is 1 at an index i, then the element of nums at i also appears in this subset. This means the solution uses a nested loop. Note that every bitset we will find is distinct, since the number it represents is also distinct.

Analysis: the time complexity of this solution is the best for this problem, at $O(2^n * n)$. The space complexity is O(1). This is because every bitset in C++ has to be preset (at compile time) with a certain number of bits. For this problem, the limit was 10, so 10 bits. So for smaller sets, indices of the bitset were offset accordingly to check only the n least significant bits. That is, since we only create bitsets representing numbers from 0 to 2^n-1, we should only check the rightmost n bits, where n <= 10.

---

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 involves the use of a recursive Depth-First-Search (DFS) through the binary tree. Recall that a BST (Binary Search Tree) is where the left subtree's nodes have values less than the root node, and the right subtree's values are always greater than the root's. Therefore, the intuition that needs to be followed is that the tree is valid only if **both** the left and right, and *all* subtrees satisfy this requiremeent, not just one of them. The steps for the algorithm:

1. We will make use of a recursive helper function to run the DFS. Its inital parameters will be the root node, a boolean to carry through the results for all subtrees below, but more importantly, the maximum and minimum values of an appropriate integer type. Those numbers are the absolute limits for the values of subtrees. If we check the right subtree, that right subtree's left nodes will also have to be checked and compared to the maximum possible limits. However, the first limit of all is a set of max and min integer values.
2. If the root is null, we can simply return true. An empty tree is always a valid BST.
3. Then, we first check the basic BST requirements for the 3-node subtree currently traversed. If they are satisfied, we traverse the left subtree accordingly, now updating the maximum or minimum node value depending on whether it is a left or right node check.
4. If the result from checking the left subtree is false, we don't need to check the right; it is already not a BST. Otherwise, we have to check the right subtree as well. The steps above explains the baisc semantics of how most of the submissions work, but it can always be made more efficient through better syntax.

Analysis: the time complexity for the above algorithm is O(N), the classic DFS through a binary tree, and so is the space complexity.

---

99. Recover Binary Search Tree

The easiest solution to this tough problem involves the use of an in-order traversal to verify and recover the tree to be a BST (Binary Search Tree). Recall that a BST is where all the left subtree's nodes are smaller than the root, and all the right subtree's nodes are larger.

---

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.

---

148. Sort List

We will sort the linked list with a special variation of merge sort, while finding middle nodes using the tortoise and hare algorithm. The algorithm runs as follows, on the given function sortList() and and helper function merge(), while given only the head node:

1. If the head is NULL or it is the only node, return head.
2. Then, we run tortoise and hare to find the middle node. This is done as follows:
    - Initialize a fast and slow node, the fast starting with head's next, and the slow initially head.
    - Then, we loop continuously until the fast node is at the end. Each loop iteration will first involve incrementing the fast node, then if fast is *still* not at the end, then we also increment slow and fast again. This makes fast run twice as "fast" as the slow.
    - In this way, when fast is at the end, slow is at the middle.
3. Once fast and slow are in place, we recursively run sortList() on both halves of the linked list. To split the linked list, there needs to be a pointer at head and the slow pointer, so that the right half starts at slow's next. To "cut" the list, we make slow's next NULL, so that when the first half is traversed in the recursive sortList(), the end is NULL.
4. In the merge function, we run the classic function from merge sort. Go through the two lists, until one of the lists is empty, and while comparing the values starting from the head, push the smaller node into the final list and increment the appropriate pointer. Finally, if the lists are of unequal length, push the remaining nodes of the longer list into the returned list. The pointer to the longer list will already contain its remaining nodes, so unlike arrays, we only need to "fuse" the nodes once.
5. Don't forget to hold a dummy at the head of a new return list for the merge() function, while traversing with a pointer initially at the same node as the dummy.

Analysis: a merge sort's time complexity is O(N*logN). The auxillary space used is O(N).

---

184. Department Highest Salary

This problem requires the use of `JOIN` and `PARTITION` statements as well as subqueries. Why? Because highest salary in *each* department is needed (done through sorting), as well as a merging with the department names based on the departmentId numbers in the employee table.

... more to come / TBC

---

185. Department Top Three Salaries

This problem requires the use of a Correlated Subquery (https://dev.mysql.com/doc/refman/8.0/en/correlated-subqueries.html) involving the two tables Employee and Department... TBC

---

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;

---

206. Reverse Linked List

This problem focuses on the fundamental practice of reversing a linked list. It is difficult to explain just by writing down, and requires the use of a whiteboard. But I'll do my best, so here are the steps to reverse a list, given the head node. The main goal is to reverse the pointer directions **in-place**. This will be a three-pointer solution.

1. Create a node `curr` to walk along through the linked list (set initially to head), a node called `next` we set to null (this will be used to track the following node), and a node called `prev` to track the new `next` pointers of the final linked list, which are initially the previous nodes.
2. Since `curr` is what guides the traversal, we loop until it becomes null pointer.
3. In each step walking through the linked list:
    - We first assign the `next` node to the next of the `curr` node to get the node to point backwards in the reversed.

    - Then, `curr` 's next node, should point to the previous node, which is initally zero and remains the previous node before `curr` for the rest of the iterations through the linked list. This reverses the pointer between the current and previous.
    - Afterwards, we prepare for the next iteration, by assigning `prev` to the current node, as that applies to the next node. We also pre-assign `curr` to the `next` node, and repeat the processes in step #3 (this one).
4. Once it is done, the next node is nullptr in the given linked list. As a result, the new head of the reversed list should be at `prev` the last node we worked with.

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

---

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.

1685. Sum of Absolute Differences in a Sorted Array

Naive Solution - easy Prefix Sum - O(1)

2881. Create a New Column

- Pandas Series...
- Column-wise operations...

2943. Maximize Area of Square Hole in Grid

Approach: This fairly difficult medium problem involves the use of a great deal of intution, some geometry, and some sorting. One should first see that the maximum square area comes from groups of consecutive bars being removed. Another important observation is that once any horizonal bars are removed, any vertical bar can be removed to increase area by the same amount, and vice versa. For example, when the first horizontal bar is removed from a grid, it doesn't matter whether you reverse the first or second vertical bar, you get the same square area. After a lot of thinking, we can conclude that the of the largest possible group of consecutive removable horizontal and vertical bars, the square of the minimum of the horizontal and vertical groups is the result.

Solution: Since the arrays of removable bars is not sorted, even though distinct, they must first be sorted. Then a helper function should be used to find only the size of the largest group of consecutive bars for both the vertical and horizontal sets. That function will work as follows: track an overall total maximum, and the last index where there were nonconsecutive bars (that is initiallly zero). While traversing the bars, update the current largest group of consecutive bars (since the last nonsecutive index), and update the return value. Once the result of this helper function for each set of bars, the answer to the problem is the minimum of the results for the two sets, squared. That is because you cannot remove more than the minimum of any of the two sets.

Analysis: the time complexity of this algorithm is O(N + N*logN), since sorting takes N*logN time, and traversing both takes O(N) time for each.