ArabicNumToChinese

/\* Here is the explanation for the code above:

1. The first part is to declare some constant arrays and a StringBuilder to store the answer.

2. The second part is the main part of the code, which is to convert the integer to Chinese number.

3. The third part is to run the code and see the results. \*/

AddSubset

/\* Here is the explanation for the code above:  
1. The result of subset of an empty set is an empty set, so we add an empty set to the result.  
2. For every element in the array, we add it to the end of every subset in the result.  
3. The subset we already have is not affected by the element we just added. So no need to add the new subset to the result. \*/

SetZeroMatrix

/\* Here is the explanation for the code above:  
1. scan the first row and column, if there is a 0, set two flags row and col. Note that, we need two flags to separate row = 0 and col = 0 cases.  
2. use first row and col to set matrix. Except the first row and col.  
3. set first row and col by using two flags. \*/

Shuffle

/\* Here is the explanation for the code above:  
1. We need to keep the original array intact, so when we reset, it must have all the original values.  
2. For shuffle, we can pick any number from the array and put it in the first position, then pick from the remaining (n-1) positions and so on. This is equivalent to picking the first position from n positions, then the second position from n-1 positions and so on.  
3. We can achieve this by picking a random number from the remaining numbers and swap it with the number in the first position, then pick a random number from the remaining (n-1) positions and swap it with the number in the second position and so on.  
4. To achieve this, we need the list of remaining numbers, so we create a list and add all the values from the array to the list. We can use the array itself as the list, but we need to remove the numbers that we have already picked. Removing from an array is an expensive process, so we use a list.  
5. We pick a random number from the remaining numbers using rand.nextInt(aux.size()). This returns a number between 0 and aux.size()-1.  
6. Once we pick a number, we swap it with the number in the current position (i) in the array. This is because we need to make sure that the number is not picked again.  
7. We remove the picked number from the list, so that it is not picked again.  
8. We repeat this process until there are no remaining numbers.  
9. The runtime complexity is O(n) because we pick n numbers from the array and each pick takes O(1) time. The space complexity is O(n) because we use a list to store the remaining numbers. \*/

SpiralMatrix

/\* Here is the explanation for the code above:  
1. Use a boolean[][] to mark the visited cells.  
2. Use a direction variable to indicate the direction we are going to move.  
3. If we are able to move to the next cell, move to the next cell.  
4. If we cannot move to the next cell, change the direction and move to the next cell.  
5. If we cannot move to the next cell and we have already visited the next cell, we know we have reached the end of the matrix. \*/

HierarchicalTraversal

/\* Here is the explanation for the code above:  
1. Create a queue and add the root node to the queue.  
2. Create a while loop that runs as long as the queue is not empty.  
3. Inside the while loop, create a new list and add it to the levels list.  
4. Create a levelLength variable that will store the current queue size.  
5. Create a for loop and run it until the levelLength variable.  
6. Inside the for loop, remove the first node from the queue and add it to the current level list.  
7. Check if the removed node has a left child. If it does, add it to the queue.  
8. Check if the removed node has a right child. If it does, add it to the queue.  
9. Add 1 to the level variable.  
10. Return the levels list. \*/

PreorderTraversal

/\* Here is the explanation for the code above:  
1. push root to stack  
2. while stack is not empty or root is not null  
 a. while root is not null  
 i. add root to list  
 ii. push root to stack  
 iii. root = root.left  
 b. root = stack.pop()  
 c. root = root.right \*/

DeleteBSTNode

//1) If key is greater than root's key, then key lies in right subtree.  
// So we recur for right subtree.  
// 2) If key is smaller than root's key, then key lies in left subtree.  
// So we recur for left subtree.  
// 3) If key is equal to root's key, then this is the node to be deleted.  
// a) Node to be deleted is leaf: Simply remove from the tree.  
// b) Node to be deleted has only one child: Copy the child to the node and delete the child  
// c) Node to be deleted has two children: Find inorder successor of the node.  
// Copy contents of the inorder successor to the node and delete the inorder successor.  
// Note that inorder predecessor can also be used. \*/

ReplaceWord

/\* Here is the explanation for the code above:  
1. The first for loop is to build a Trie Tree. (I'm not gonna explain the Trie Tree here since it's not the point for this question)  
2. The second for loop is to find the shortest root for each word.  
3. The time complexity depends on the length of the sentence O(n) and the number of roots we have O(m), which is O(n\*m).  
4. The space complexity is the size of the Trie Tree which is O(26\*n) = O(n). \*/

Dijkstra

///\* Here is the explanation for the code above:  
//1. 在输入中，第一行的两个整数分别表示图的高和宽，第三行的两个整数表示起点的坐标，后面的每行表示图中的一个点，对应的数字表示该点的值。  
//2.在solve函数中，首先初始化了isVisited和d数组，isVisited用来标记某一点是否被访问过，d用来记录起点到某一点的最短距离。  
//3.将起点标记为已访问，距离初始化为0。  
//4.进入while循环，当所有点都被访问过或者当前最小距离为MAX时跳出循环，否则继续执行。  
//5.在循环中，首先找到d中值最小的点，将其标记为已访问，然后更新起点到其他点的距离。  
//6.最后返回最短距离。 \*/

maxArea

/\* Here is the explanation for the code above:  
1. We take two pointers, one at the beginning and one at the end of the array constituting the length of the lines.  
2. Futher, we maintain a variable maxareamaxarea to store the maximum area obtained till now. At every step, we find out the area formed between them, update maxareamaxarea and move the pointer pointing to the shorter line towards the other end by one step.  
3. Initially, we consider the area constituting the exterior most lines. Now, to maximize the area, we need to consider the area between the lines of larger lengths. If we try to move the pointer at the longer line inwards, we won't gain any increase in area, since it is limited by the shorter line.  
4. But moving the shorter line's pointer could turn out to be beneficial, as per the same argument, despite the reduction in the width. This is done since a relatively longer line obtained by moving the shorter line's pointer might overcome the reduction in area caused by the width reduction. \*/

MaxProfit

/\* Here is the explanation for the code above:  
1. At the very beginning, we set the min price to be a very large number and set the max profit to be 0.  
2. Then we iterate through the array, and each time we find a new minimum price, we update the min price.  
3. For each price, we calculate the profit by substracting the min price from the current price.  
4. Then we update the max profit if the current profit is larger than the previous max profit.  
5. Finally, we return the max profit. \*/

MinSubArray

/\* Here is the explanation for the code above:  
1. use two pointers, i and k, to mark the start and end of the subarray  
2. use a variable sum to record the sum of the subarray between i and k  
3. if sum is smaller than s, then we need to add another element to the subarray, so we move k one step forward and add the value of nums[k] to sum  
4. if sum is larger than s, then we need to remove one element from the subarray, so we move i one step forward and subtract the value of nums[i] from sum  
5. after each step, we need to update the value of min, which records the minimum length of the subarray whose sum is larger than or equal to s  
6. the loop will terminate when k reaches the end of the array and i reaches the end of the array  
7. we need to return 0 if min is still Integer.MAX\_VALUE after the loop, which means there is no subarray whose sum is larger than or equal to s \*/

Rain

/\* Here is the explanation for the code above:  
1. For each element in the array, we find the maximum level of water it can trap after the rain,  
 which is equal to the minimum of maximum height of bars on both the sides minus its own height.  
2. We then sum up the values obtained for all the elements in the array. \*/

RemoveDuplicates

/\* Here is the explanation for the code above:  
1. Use a variable (c) to keep track of the current position.  
2. Loop through the array from 1 to the end.  
3. If the current value is not equal to the value at c, increment c and set the value at c to the current value.  
4. Return c + 1. \*/

ThreeNumSum

/\* Here is the explanation for the code above:  
1. Sort the array, so that we can use two pointers to scan the array.  
2. We need to skip the duplicates to avoid duplicate triplets.  
3. For each element, we use two pointers to scan the rest of the array. If the sum of three elements is 0, then add it to the result. Otherwise, if the sum is bigger than 0, then move the right pointer to left; if the sum is smaller than 0, then move the left pointer to right. \*/

All1SquareMatrix

/\* Here is the explanation for the code above:  
1. matrix[i][j] == 0, dp[i][j] = 0  
2. matrix[i][j] == 1, dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1  
3. the total number of squares is the sum of all elements in dp. \*/

MaxSubString

/\* Here is the explanation for the code above:  
1. Find the longest common subsequence of two strings.  
2. Use the length of longest common subsequence to get the shortest common supersequence.  
3. The length of the shortest common supersequence = (length of text1) + (length of text2) - (length of longest common subsequence) \*/

MinStairs

/\* Here is the explanation for the code above:  
1. f1, f2 are the minimum cost to climb to the top starting from index i+1, i+2 respectively.  
2. The goal is to get to the top from index -1, so the final answer is Math.min(f1, f2).  
3. At each step, we maintain f1 = f(i+1), f2 = f(i+2).  
4. f(i) = cost[i] + Math.min(f(i+1), f(i+2)).  
5. So, the code traverses backward. \*/

YHTriangle

/\* Here is the explanation for the code above:  
1. The first row is always [1]. Initialize a list, and add that to the triangle list.  
2. The second row is always [1,1]. We can hard code this. Add that to the triangle list.  
3. Now, we have to generate the third row.  
The third row is [1,2,1]. The second row is [1,1].  
We can generate the third row by adding the second row shifted left, to itself shifted right.  
Specifically, [1,2,1] = [0+1,1+1,1+0]. We prepend and append a 0.  
4. Now, we have to generate the fourth row.  
Add a 0 in the beginning and at the end of the third row. [1,2,1] -> [0,1,2,1,0].  
Add that to the triangle list.  
5. Continue this process until you have generated the required number of rows. \*/

CircleList

/\* Here is the explanation for the code above:  
1. If there is no cycle, the fast pointer will stop at the end of the linkedlist.  
2. If there is a cycle, the fast pointer will eventually meet with the slow pointer.  
3. If there is a cycle, the fast pointer will eventually meet with the slow pointer. \*/

HappyNum

/\* Here is the explanation for the code above:  
1. First of all, let's suppose there is a cycle in this sequence, and the cycle starts with number x, ends with number y. We use function f(n) to calculate the next number, f(f(f...(f(n)))) = y.  
2. There are two pointers in this sequence, one goes one step each time, another one goes two steps each time. So when they meet, we know there is a cycle. Let's suppose the length of the cycle is k.  
3. When the two pointers meet, the slow one goes k steps. At this time, the fast one goes 2k steps. Since the fast one goes two steps each time, and it already goes k steps, so the number of steps it needs to go to reach y is (k - (2k - k)) = k.  
4. f(f(f...(f(y)))) = y, which means the next number of y is y itself, which means the number of steps it needs to go to reach y is 1. So we know the slow pointer needs to go k more steps after they meet. Since the slow pointer already goes k steps, so the total steps it goes to reach y is 2k, which means the total steps it goes to reach y is the length of the cycle.  
5. We already know the length of the cycle is k, so we can conclude that the slow pointer goes k steps to reach y, and go k steps to reach x. Since the total steps it goes is 2k, we can conclude that the slow pointer goes k steps to reach x. So the next number of x is x itself.  
6. Now we have proved that if there is a cycle, then there must be a number x, the next number of x is x itself. So if the number is not a happy number, then it must contains some numbers which are visited twice since there are a cycle. So we can use this conclusion to solve this problem. \*/

MiddleNode

/\* Here is the explanation for the code above:  
1. we have a slow and fast pointer starting from head.  
2. we move slow pointer one step at a time and fast pointer two steps at a time.  
3. when fast reaches the end, slow will be at the middle. \*/

GlassBall

//\* dp[i] = min(dp[i-j]+1, dp[j]) (0 < j < i)  
// \*  
// \* dp[i] means the min number of squares that can consist of sum i  
// \*  
// \* dp[i-j]+1 means the min number of squares that can consist of sum j plus one square with length i-j  
// \*  
// \* dp[j] means the min number of squares that can consist of sum j  
// \*  
// \* So dp[i-j]+1 means one square with length i-j and dp[j] means one square with length j  
// \*  
// \* So dp[i] means one square with length i and dp[j] plus one square with length i-j  
// \*  
// \* So we just need to find the min number of dp[i-j]+1 and dp[j] for all j < i  
// \*  
// \* For example, if i is 5, we just need to find the min number of dp[4]+1 and dp[3]+1  
// \*  
// \* So the final dp[5] will be the min number of all the min number of dp[4]+1 and dp[3]+1  
// \*  
// \* So dp[i] = min(dp[i-j]+1, dp[j]) (0 < j < i)  
// \* \*/

FirstUniqueChar

//\* input: a string  
//\* output: the index of the first unique character  
//\* method: count the number of characters in the string, and then find the first character that only appears once

//This code first creates an empty HashMap. Then, it iterates over the elements of the array using a for loop. For each element of the array, it checks if the element is already present in the HashMap. If the element is present, it means that we have found a pair of elements whose sum is equal to the target value. If the element is not present, we put the element of the array in the HashMap. The time complexity of this approach is O(n) because we are iterating the array only once.

SumOf2Num

//This code first creates an empty HashMap. Then, it iterates over the elements of the array using a for loop. For each element of the array, it checks if the element is already present in the HashMap. If the element is present, it means that we have found a pair of elements whose sum is equal to the target value. If the element is not present, we put the element of the array in the HashMap. The time complexity of this approach is O(n) because we are iterating the array only once.

FindKthLargest

/\* Here is the explanation for the code above:  
1. First, we insert all the elements into a heap. The size of the heap is always maintained at k.  
2. If the size of the heap is greater than k, then we remove the top element from the heap.  
This is because, we are interested in the kth largest element and hence if it is present in the heap,  
it has to be among the top k elements in the heap.  
3. Finally, the top element in the heap is the kth largest element. \*/

Add2Nums

/\* Here is the explanation for the code above:  
1. I create a new ListNode to store the sum of the 1st digit of l1 and l2.  
2. I iterate through the rest of the list while l1 or l2 is not null.  
3. When l1 or l2 is null, I will set a or b to 0 respectively.  
4. I add a and b and store it in t.  
5. I create a new ListNode of t and add it to the end of the list.  
6. I iterate through the list and handle the carry.  
7. If the last digit is 10, I create a new ListNode of 0 and add it to the end of the list.  
8. I return the list. \*/

CopyRandomList

/\* Here is the explanation for the code above:  
1. If we encounter a node which is already visited then we return the clone of it.  
2. We create a copy of the given node and put it in the visited dictionary.  
3. Recursively copy the remaining linked list starting once from the next pointer and then from the random pointer.  
4. Now we update the next and random pointers for the clones using the visited dictionary. \*/

DeleteNNodeFromEnd

/\* Here is the explanation for the code above:  
1. We use two pointers to locate the position of the node to be deleted.  
2. We need to use dummy node to handle the case when the head node is removed.  
3. The second pointer is always n+1 nodes behind the first pointer. \*/

GetIntersectionNode

/\* Here is the explanation for the code above:  
1. Traverse list A and store the address / reference to each node in a hash set.  
2. Then check every node bi in list B: if bi appears in the hash set, then bi is the intersection node.  
Complexity Analysis  
Time complexity : O(m+n)O(m+n).  
Space complexity : O(m)O(m) or O(n)O(n). \*/

ReverseList

/\* Here is the explanation for the code above:  
1. First, we define a pre pointer, which is null at the beginning.  
2. Then, we define a temp pointer, which points to head at the beginning.  
3. Then, we start a while loop. The loop runs when temp is not null.  
4. In the loop, we define a new node t, which points to temp.next.  
5. Then, we set temp.next to pre.  
6. Then, we set pre to temp.  
7. Then, we set temp to t.  
8. Then, we continue the loop.  
9. Finally, we return pre. \*/

RotateList

/\* Here is the explanation for the code above:  
1. Get the length of the list;  
2. Move to the (l - k % l)th node;  
3. Put the tail node to the head and cut the connection between the tail and the (l - k % l - 1)th node. \*/

Merge

/\* Here is the explanation for the code above:  
1. Sort the intervals based on their start time  
2. Add the first interval to the output array  
3. Iterate through the sorted array and check if the end time of the last interval is greater than the start time of the new interval  
4. If not, we add the new interval to the output array and update the last interval  
5. If yes, then we update the end time of the last interval to be the max of the end time of the last interval and the end time of the new interval  
6. Return the output array \*/

IntReverse

/\* Here is the explanation for the code above:  
1. Get the last digit of the number: y = x % 10.  
2. Update the last digit of the reverse: rs = rs \* 10 + y.  
3. Remove the last digit of the number: x = x / 10.  
4. Repeat steps 1. to 3. until x is no longer greater than zero.  
5. Check overflow/underflow condition: if (rs \* 10 / 10 != rs) return 0; \*/

NumOfReplies

/\* Here is the explanation for the code above:  
1. Negative numbers are not palindrome, for example -123 is not a palindrome since the '-' does not equal to '3'. So we can return false for negative numbers.  
2. Now let's think about how to revert the last half of the number. For number 1221, if we do 1221 % 10, we get the last digit 1, to get the second to the last digit, we need to remove the last digit from 1221, we could do so by dividing it by 10, 1221 / 10 = 122. Then we can get the last digit again by doing a modulus by 10, 122 % 10 = 2, and if we multiply the last digit by 10 and add the second last digit, 1 \* 10 + 2 = 12, it gives us the reverted number we want. Continuing this process would give us the reverted number with more digits.  
Now the question is, how do we know that we've reached the half of the number?  
Since we divided the number by 10, and multiplied the reversed number by 10, when the original number is less than the reversed number, it means we've processed half of the number digits. \*/

StringMultiplication

/\* Here is the explanation for the code above:  
1. The product of two numbers cannot exceed the sum of the two lengths. (e.g. 99 \* 99 cannot be five digit)  
2. num1[i] \* num2[j]` will be placed at indices `[i + j`, `i + j + 1]`  
3. We initialize our answer array with `0`s and fill it from the right  
4. Similar to how we would do multiplication on paper, for each `i` in `num1` and `j` in `num2`  
5. We get the current product from `num1[i] \* num2[j]` (plus the carry from the previous index)  
6. The current digit will be `products % 10` and the carry will be `products / 10`  
7. We then update the answer at `i + j` (the first index) and `i + j + 1` (second index) with the new digit  
8. After the two loops, we remove any leading `0`s from our answer  
9. If the entire answer array is `0`s, we return 0, otherwise we return the string built from the answer array \*/

StringToInt  
 /\* Here is the explanation for the code above:  
1. null or empty string  
2. white spaces  
3. +/- sign  
4. calculate real value  
5. handle min & max \*/

TrailingZeros

/\* Here is the explanation for the code above:  
1. First, let's see how the number of 0s is generated.  
If we write down all the numbers from 1 to n, and count the number of 0s in each column, we'll find that the number of 0s at each column is [n/5] + [n/5^2] + [n/5^3] + ....  
For example, the number of 0s from 1 to 105 is 20, from 1 to 1010 is 249.  
The reason that we use [n/5] + [n/5^2] + [n/5^3] + ... to compute the number of 0s at each column is as follows:  
 1) [n/5] is the number of numbers from 1 to n that are multiples of 5.  
 2) [n/5^2] is the number of numbers from 1 to n that are multiples of 5^2 and not multiples of 5.  
 3) [n/5^3] is the number of numbers from 1 to n that are multiples of 5^3 and not multiples of 5^2.  
 ... and so on.  
This way, we can count the number of 0s in O(logn) time.  
  
2. Then, let's see how the number of 5s is generated.  
If we write down all the numbers from 1 to n, and count the number of numbers that are multiples of 5, we'll find that the number of 5s at each column is [n/5] + [n/5^2] + [n/5^3] + ....  
For example, the number of 5s from 1 to 105 is 25, from 1 to 1010 is 252.  
The reason that we use [n/5] + [n/5^2] + [n/5^3] + ... to compute the number of 5s at each column is as follows:  
 1) [n/5] is the number of numbers from 1 to n that are multiples of 5.  
 2) [n/5^2] is the number of numbers from 1 to n that are multiples of 5^2.  
 3) [n/5^3] is the number of numbers from 1 to n that are multiples of 5^3.  
 ... and so on.  
This way, we can count the number of 5s in O(logn) time.  
  
3. Finally, let's see how the number of 2s is generated.  
If we write down all the numbers from 1 to n, and count the number of numbers that are multiples of 2, we'll find that the number of 2s at each column is [n/2] + [n/2^2] + [n/2^3] + ....  
For example, the number of 2s from 1 to 105 is 52, from 1 to 1010 is 504. \*/

MergeLists

/\* Here is the explanation for the code above:  
1. We need to define a head node, and a pre node which points to the head node. The reason is that we need to modify the linked list, but we still need to return the head node.  
2. We need to compare the first element of l2 with the first element of the linked list. If l2.val is smaller than the first element of the linked list, we need to insert l2 to the linked list. Otherwise, we need to find the proper position to insert l2. If we find the end of the linked list, then we need to insert l2 to the end of the linked list.  
3. We need to move the pre node to the next position.  
4. We need to move the l2 node to the next position. \*/

SearchSpanSortedArray

/\* Here is the explanation for the code above:  
1. Find middle point mid = (l + h)/2  
2. If key is present at middle point, return mid.  
3. Else If arr[l..mid] is sorted  
 a) If key to be searched lies in range from arr[l]  
 to arr[mid], recur for arr[l..mid].  
 b) Else recur for arr[mid+1..h]  
4. Else (arr[mid+1..h] must be sorted)  
 a) If key to be searched lies in range from arr[mid+1]  
 to arr[h], recur for arr[mid+1..h].  
 b) Else recur for arr[l..mid] \*/

InvertTree

/\* Here is the explanation for the code above:  
1. for any node, we swap its left and right child.  
2. for the left child, we swap its left and right child  
3. for the right child, we swap its left and right child  
  
We can use a stack to store all the left and right child of a node.  
For example, we have a tree like this:  
 1  
 / \  
 2 3  
 / \ / \  
 4 5 6 7  
 / \  
 8 9  
  
Initially, we push 1 in the stack. The stack is [1]. Then we pop 1 out of the stack and swap its left and right child. The tree becomes:  
 1  
 / \  
 3 2  
 / \ / \  
 6 7 4 5  
 / \  
 8 9  
We push 2 and 3 in the stack. The stack is [2, 3]. Then we pop 3 out of the stack and swap its left and right child. The tree becomes:  
 1  
 / \  
 3 2  
 / \ / \  
 7 6 4 5  
 / \  
 8 9  
We push 6 and 7 in the stack. The stack is [2, 6, 7]. Then we pop 7 out of the stack and swap its left and right child. The tree becomes:  
 1  
 / \  
 3 2  
 / \ / \  
 7 6 4 5  
 / \  
 9 8  
We push 4, 5, 6 in the stack. The stack is [2, 6, 5, 4]. Then we pop 4 out of the stack and swap its left and right child. The tree becomes:  
 1  
 / \  
 3 2  
 / \ / \  
 7 6 5 4  
 / \  
 9 8  
We push 8, 9 in the stack. The stack is [2, 6, 5, 9, 8]. Then we pop 8 out of the stack and swap its left and right child. The tree becomes:  
 1  
 / \  
 3 2  
 / \ / \  
 7 6 5 4  
 / \  
 9 8  
  
We push 9 in the stack. The stack is [2, 6, 5, 9]. Then we pop 9 out of the stack and swap its left and right child. The tree becomes:  
 1  
 / \  
 3 2  
 / \ / \  
 7 6 5 4  
 / \  
 8 9  
The stack is empty now, so the algorithm ends here. \*/

LowestCommonAncestor

/\* Here is the explanation for the code above:  
1. if root is null, then it means we have reached the end of the tree and we return null  
2. if root is either p or q, then we return root because we know that the other node is in the subtree of this root  
3. if root is neither p nor q, then we recursively search for p and q in the left and right subtree  
4. if both left and right are not null, then it means that p and q are in different subtrees, so we return root  
5. if either left or right is null, then it means that both p and q are in the same subtree, so we return the one that is not null \*/

MaxDepth

/\* Here is the explanation for the code above:  
1. We need to find the depth of the tree, so we need to return the max value of the depth of the left subtree and the right subtree.  
2. We use recursion to find the depth of the tree.  
3. The recursion has a base case, which is when the root is null, we return 0.  
4. Otherwise, we calculate the depth of the left subtree and the right subtree, and return the max value of them.  
5. In the recursion, we use a variable to record the depth of the left subtree and the right subtree, and we need to add them by 1, because the root is not null.  
6. At last, we return the max value of the depth of the left subtree and the right subtree plus 1. \*/

Merge2Lists

/\* Here is the explanation for the code above:  
1. The base cases are when either list is null. Then there's nothing to merge, so you just return the other non-null list.  
2. Otherwise, you compare the heads of the two lists, and add the smaller one to the merged list. The next element of the merged list is then set to the merge of the lists that didn't contribute the head.  
3. Finally, you return the merged list. \*/

RemoveLeafNodes

/\* Here is the explanation for the code above:  
1. We need to go to the bottom of the tree first, then go up to the root.  
2. So we use post-order traversal.  
3. When we reach a leaf node, we check whether its value is equal to target.  
4. If it is equal to target, we return null, which means that this node should be removed.  
5. If it is not equal to target, we return this node.  
6. When we return a non-null value, we need to assign it to the left or right child of its parent node.  
7. When we assign the value to the left or right child of its parent node, we need to check whether the left or right child is null.  
8. If the left or right child is null, we need to assign null to the left or right child of the parent node.  
9. Otherwise, we do nothing.  
10. After we traverse all the nodes, we will return the root node of the tree. \*/

SymmetricBinaryTree

/\* Here is the explanation for the code above:  
1. Initially, the algorithm checks if the root node is null. If so, then we conclude that the tree is symmetric. Otherwise, we call the recursive helper function isMirror.  
2. The isMirror function takes two TreeNode objects as arguments. The function checks if the two TreeNode objects are null. If so, the function returns true. If not, the function checks if the values of the TreeNode objects are equal. If not, the function returns false. Otherwise, the function returns the result of a recursive call to isMirror. The two recursive calls are made with the following arguments: the left subtree of the left TreeNode object and the right subtree of the right TreeNode object; the right subtree of the left TreeNode object and the left subtree of the right TreeNode object. \*/

Brackets

/\* Here is the explanation for the code above:  
1. As we can see the recursion tree, we can think about the base case. In this case, it is when the length of the string is equal to 2\*n.  
2. Then we can think about the recursion rule. In this case, it is when to add "(" or ")".  
3. Then we can think about the condition to add "(". In this case, it is when the number of "(" is smaller than n.  
4. Then we can think about the condition to add ")". In this case, it is when the number of "(" is larger than the number of ")". \*/

CombinationSum

/\* Here is the explanation for the code above:  
1. Sort the candidates;  
2. Use HashSet to avoid duplicate result;  
3. Use start to avoid duplicate result;  
4. Use target < 0 to avoid unnecessary computation. \*/

RegularExpressionMatching

/\* Here is the explanation for the code above:  
1. If a star is present in the pattern, it will be in the second position pattern[1].  
Then, we may ignore this part of the pattern, or delete a matching character in the text.  
If we have a match on the remaining strings after any of these operations, then the initial inputs matched.  
  
2. If the pattern[1] is not a star, then we must match the current character of the text to  
that of the pattern. Again, if there is a match, we proceed. Otherwise, we return false.  
  
The base cases of the recursion are when the length of the pattern is 0 or when the length  
of the pattern is 1 and isn't followed by a star. In both of these cases, we simply check  
if the lengths of the text and pattern are equal. If they are, we have a match on our hands! \*/

Calculator

/\* Here is the explanation for the code above:  
1. It uses two stack to store the numbers and operations, respectively.  
2. It uses a variable temp to store the number which is not splited by space.  
3. It uses a variable n to store the length of the input string.  
4. For the for loop, it checks if the current character is a space. If it is, continue the loop. Otherwise, it checks if the current character is a number. If it is, it use the variable temp to store the number. Otherwise, it checks if the current character is an operation. If it is, it checks the top operation of the operation stack. If the top operation is a left parenthesis, it will push the current operation into the operation stack. Otherwise, it will pop the top operation of the operation stack, and pop two number from the number stack, and calculate the result and push the result into the number stack. It will do the operation until the top operation of the operation stack is a left parenthesis. Then it will push the current operation into the operation stack. Otherwise, it checks if the current character is a left parenthesis. If it is, it will push the current operation into the operation stack. Otherwise, it will pop the top operation of the operation stack. If the operation is a left parenthesis, it will stop popping. Otherwise, it will pop two number from the number stack, and calculate the result and push the result into the number stack. It will do the operation until the top operation of the operation stack is a left parenthesis. Then it will pop the top operation of the operation stack.  
5. After the for loop, it will check if the variable temp is not -1. If it is not -1, it will push the temp into the number stack.  
6. After that, it will check if the operation stack is not empty. If it is not empty, it will pop the top operation of the operation stack. Then it will pop two number from the number stack, and calculate the result and push the result into the number stack. It will do the operation until the operation stack is empty.  
7. Finally, it will return the top element of the number stack. \*/

Calculator2

/\* Here is the explanation for the code above:  
1. The variable n stores the number of consecutive digits.  
2. The variable operand stores the operand that is being built from consecutive digits.  
3. The stack stores the operands and the signs. It is a stack of Objects.  
 When we see a number, we keep pushing it to the stack.  
 When we see a sign, we pop one number from the stack and do the calculation.  
 Therefore, when we see a sign, the top of the stack is always a number.  
4. The function evaluateExpr evaluates the expression value until the next closing bracket.  
 The idea is as follows:  
 - We keep a stack of numbers (integer) and signs (character) for each opening bracket.  
 - When we see a closing bracket, we start popping the numbers and signs from the stack,  
 until we reach the corresponding opening bracket.  
 We then get the sum of the popped numbers, update the stack with the new sum,  
 and continue to calculate the rest of the expression.  
5. We use a while loop to iterate over the expression.  
 For each character, there are two cases:  
 - The character is a digit:  
 We update the operand: operand = operand \* 10 + (int) ch - '0'.  
 - The character is a sign (‘+’ or ‘-‘) or a closing bracket ‘)’:  
 We evaluate the expression to the next closing bracket,  
 by calling the function evaluateExpr.  
 We then update the result and the sign:  
 - res += sign \* operand, where sign = 1 if the sign is ‘+’ and sign = -1 if the sign is ‘-‘.  
 - If the character is ‘)’, we break the while loop.  
6. Finally, we return the result. \*/

LongestValidBrackets

/\* Here is the explanation for the code above:  
1. Push -1 into the stack to mark the end of the previous valid substring.  
2. Iterate over the string, if current character is '(', push index into stack.  
3. If current character is ')', pop from stack. If stack is not empty, then update the max length, current index minus top of stack.  
4. If stack is empty, push current index into stack to mark the end of the current valid substring. \*/

MaxQueue

/\* Here is the explanation for the code above:  
1. The queue is the original queue, which stores the elements.  
2. The max is the queue that stores the max value of the original queue.  
3. The max queue is a monotonic queue, which means the elements in the queue are in ascending order.  
4. When we push an element to the original queue, we need to find the right position for it in the max queue. So we should pop the elements in the max queue from the end until the last element is larger than the element we are going to push.  
5. When we pop an element from the original queue, we need to check whether the element is the largest element in the max queue. If it is, we should pop it from the max queue. \*/

MinStack

/\* Here is the explanation for the code above:  
1. If the stack is empty, push the number into the stack, and push the number again into the stack.  
2. If the stack is not empty, compare the number with the top of the stack, if it is greater than the top, push the top again into the stack. If it is smaller, push the number into the stack.  
3. When pop the element, pop twice. The first pop is to pop the top of the stack, the second one is to pop the min element.  
4. When peek the element, the second top element is the top element in the stack. \*/

RemoveDuplicateLetters

/\* Here is the explanation for the code above:  
1. scan the string from left to right, use stack to store the characters,  
2. for each character, if it is already in the stack, skip it;  
3. if the current character is smaller than the top character of the stack,  
and the top character of the stack exists in the rest of the string,  
pop the stack to remove the top character,  
4. push the current character to the stack.  
5. after scanning, pop the stack to form the result string. \*/

ValidBrackets

/\* Here is the explanation for the code above:  
1. Push character into stack if it's open parentheses.  
2. If it's close parentheses, pop from stack and compare with it.  
3. If not same, return false.  
4. If stack is empty, return true.  
5. Else return false. \*/

DelineateLetterRange

/\* Here is the explanation for the code above:  
1. We need an array to store the last index of each char;  
2. We need one variable to store the end index of the current sub string;  
3. We need one variable to store the start index of the current sub string;  
4. We need one arraylist to store the length of each partition.  
  
The loop is to traverse the string and update the end index of the current partition. If i == end, then we get one partition. \*/

ZTransfer

/\* Here is the explanation for the code above:  
1. We need to consider the special cases that numRows = 1 and numRows = 2.  
2. We need to find the relationship between the index of the characters in the original string and the index of the characters in the ZigZag string.  
3. For the first and the last row, the characters' indexes in the original string are arithmetic sequence with the common difference of 2 \* (numRows - 1).  
4. For the other rows, the characters' indexes in the original string are arithmetic sequence with the common difference of 2 \* (numRows - 1 - i) and 2 \* i, respectively.  
5. We can find the corresponding characters in the original string according to the indexes in the ZigZag string and append them to the ZigZag string. \*/

ProducerConsumer

///\* Here is the explanation for the code above:  
//1. The lock is initialized to a ReentrantLock object, and the condition variable is initialized to a condition object of the lock, which is equivalent to creating a queue of threads waiting for the condition variable.  
//2. Producer and Consumer implement the Runnable interface, and the Producer and Consumer threads are created by creating two threads.  
//3. When the producer produces, first judge whether the queue is full, if it is full, wait for the consumer to consume, then release the lock, and then wait for the consumer to consume, if it is not full, then the producer produces, count++, and then signal the consumer to consume.  
//4. When the consumer consumes, first judge whether the queue is empty. If it is empty, wait for the producer to produce, then release the lock, and then wait for the producer to produce. If it is not empty, then the consumer consumes, count--, and then signal the producer to produce.  
//5. The main function creates 5 producers and 5 consumers, which are executed in sequence according to the order of the threads. \*/

TreeTraversal

//\* 1. put root in stack  
// \* 2. if stack is not empty, pop one node and print its value  
// \* 3. push its right child, if it has one  
// \* 4. push its left child, if it has one  
// \* 5. repeat 2-4 \*/