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. \*/

Cons:有废话\*5

Pros:清楚\*5，有解释的\*5

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。 \*/

Cons:需要额外知识\*5，不够细节\*3

Pros:逻辑\*5，清楚\*5

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. \*/

Cons:缺少背景\*2，缺少上下文\*3

Pros:逻辑\*5清楚\*5,理解\*5，简洁\*2

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. \*/

Cons:缺少背景\*2，对新手不友好\*3

Pros:很详细\*3，清楚\*3

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. \*/

Cons:没背景\*1

Pros:清楚\*5，有逻辑\*5

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 \*/

Cons:没背景\*1

Pros:有逻辑\*5，细节\*2，清晰\*2，简洁\*5

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. \*/

Cons:没背景\*1

Pros:有逻辑\*5，细节\*2，清晰\*2

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. \*/

Cons:少细节\*3，没背景\*1，需要前置知识\*3

Pros:有逻辑\*4，清晰\*2

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. \*/

Cons:没例子\*1，没上下文\*3

Pros:有逻辑\*5，细节\*5，清晰\*5，Comprehensive\*2

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. \*/

Cons:没逻辑\*5，没细节\*5

Pros:无