**Bloomberg**

Introduction

Top interview questions asked by Bloomberg as voted by the community.

We compile this list thoroughly so you can save time and get well-prepared for a Bloomberg interview.

Completing this card should give you a good idea of the type of questions you would encounter in your Bloomberg interview.

Array and Strings

 Two Sum

 Longest Substring Without Repeating Characters

 3Sum

 Valid Anagram

 Spiral Matrix

 Merge Sorted Array

 Pascal's Triangle

 Integer to English Words

 Move Zeroes

 First Unique Character in a String

**String Compression**

 Subarray Sum Equals K

**Candy Crush**

Linked List

 Add Two Numbers

 Merge Two Sorted Lists

**Reverse Linked List II**

 Copy List with Random Pointer

 Reverse Linked List

 Add Two Numbers II

Trees and Graphs

 Validate Binary Search Tree

 Construct Binary Tree from Preorder and Inorder Traversal

 Flatten Binary Tree to Linked List

 Populating Next Right Pointers in Each Node

 Populating Next Right Pointers in Each Node II

 Binary Tree Right Side View

 Number of Islands

 Kth Smallest Element in a BST

 Alien Dictionary

 Decode String

 Flatten a Multilevel Doubly Linked List

Heap, Queue, Stack

 Valid Parentheses

 Trapping Rain Water

 Sliding Window Maximum

 Meeting Rooms II

Sorting and Searching

 Median of Two Sorted Arrays

 Search in Rotated Sorted Array

 Merge Intervals

**Sort Characters By Frequency**

 Top K Frequent Words

Dynamic Programming

 Longest Palindromic Substring

 Maximum Subarray

 Word Break

Design

 LRU Cache

 Moving Average from Data Stream

Others

 Reverse Integer

 Word Search

**String Compression**

Given an array of characters chars, compress it using the following algorithm:

Begin with an empty string s. For each group of **consecutive repeating characters** in chars:

* If the group's length is 1, append the character to s.
* Otherwise, append the character followed by the group's length.

The compressed string s **should not be returned separately**, but instead be stored **in the input character array chars**. Note that group lengths that are 10 or longer will be split into multiple characters in chars.

After you are done **modifying the input array**, return *the new length of the array*.

**Follow up:**  
Could you solve it using only O(1) extra space?

**Example 1:**

**Input:** chars = ["a","a","b","b","c","c","c"]

**Output:** Return 6, and the first 6 characters of the input array should be: ["a","2","b","2","c","3"]

**Explanation:** The groups are "aa", "bb", and "ccc". This compresses to "a2b2c3".

**Example 2:**

**Input:** chars = ["a"]

**Output:** Return 1, and the first character of the input array should be: ["a"]

**Explanation:** The only group is "a", which remains uncompressed since it's a single character.

**Example 3:**

**Input:** chars = ["a","b","b","b","b","b","b","b","b","b","b","b","b"]

**Output:** Return 4, and the first 4 characters of the input array should be: ["a","b","1","2"].

**Explanation:** The groups are "a" and "bbbbbbbbbbbb". This compresses to "ab12".

**Example 4:**

**Input:** chars = ["a","a","a","b","b","a","a"]

**Output:** Return 6, and the first 6 characters of the input array should be: ["a","3","b","2","a","2"].

**Explanation:** The groups are "aaa", "bb", and "aa". This compresses to "a3b2a2". Note that each group is independent even if two groups have the same character.

**Constraints:**

* 1 <= chars.length <= 2000
* chars[i] is a lower-case English letter, upper-case English letter, digit, or symbol.

How do you know if you are at the end of a consecutive group of characters?

#### **Approach #1: Read and Write Heads [Accepted]**

**Intuition**

We will use separate pointers read and write to mark where we are reading and writing from. Both operations will be done left to right alternately: we will read a contiguous group of characters, then write the compressed version to the array. At the end, the position of the write head will be the length of the answer that was written.

**Algorithm**

Let's maintain anchor, the start position of the contiguous group of characters we are currently reading.

Now, let's read from left to right. We know that we must be at the end of the block when we are at the last character, or when the next character is different from the current character.

When we are at the end of a group, we will write the result of that group down using our write head. chars[anchor] will be the correct character, and the length (if greater than 1) will be read - anchor + 1. We will write the digits of that number to the array.

|  |
| --- |
| class Solution {  public int compress(char[] chars) {  int anchor = 0, write = 0;  for (int read = 0; read < chars.length; read++) {  if (read + 1 == chars.length || chars[read + 1] != chars[read]) {  chars[write++] = chars[anchor];  if (read > anchor) {  for (char c: ("" + (read - anchor + 1)).toCharArray()) {  chars[write++] = c;  }  }  anchor = read + 1;  }  }  return write;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of chars.
* Space Complexity: O(1)*O*(1), the space used by read, write, and anchor.

**Candy Crush**

This question is about implementing a basic elimination algorithm for Candy Crush.

Given a 2D integer array board representing the grid of candy, different positive integers board[i][j] represent different types of candies. A value of board[i][j] = 0 represents that the cell at position (i, j) is empty. The given board represents the state of the game following the player's move. Now, you need to restore the board to a *stable state* by crushing candies according to the following rules:

1. If three or more candies of the same type are adjacent vertically or horizontally, "crush" them all at the same time - these positions become empty.
2. After crushing all candies simultaneously, if an empty space on the board has candies on top of itself, then these candies will drop until they hit a candy or bottom at the same time. (No new candies will drop outside the top boundary.)
3. After the above steps, there may exist more candies that can be crushed. If so, you need to repeat the above steps.
4. If there does not exist more candies that can be crushed (ie. the board is *stable*), then return the current board.

You need to perform the above rules until the board becomes stable, then return the current board.

**Example:**

**Input:**

board =

[[110,5,112,113,114],[210,211,5,213,214],[310,311,3,313,314],[410,411,412,5,414],[5,1,512,3,3],[610,4,1,613,614],[710,1,2,713,714],[810,1,2,1,1],[1,1,2,2,2],[4,1,4,4,1014]]

**Output:**

[[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[110,0,0,0,114],[210,0,0,0,214],[310,0,0,113,314],[410,0,0,213,414],[610,211,112,313,614],[710,311,412,613,714],[810,411,512,713,1014]]

**Explanation:**



**Note:**

1. The length of board will be in the range [3, 50].
2. The length of board[i] will be in the range [3, 50].
3. Each board[i][j] will initially start as an integer in the range [1, 2000].

#### **Approach #1: Ad-Hoc**

**Intuition**

We need to simply perform the algorithm as described. It consists of two major steps: a crush step, and a gravity step. We work through each step individually.

**Algorithm**

Crushing Step

When crushing, one difficulty is that we might accidentally crush candy that is part of another row. For example, if the board is:

123

145

111

and we crush the vertical row of 1s early, we may not see there was also a horizontal row.

To remedy this, we should flag candy that should be crushed first. We could use an auxillary toCrush boolean array, or we could mark it directly on the board by making the entry negative (ie. board[i][j] = -Math.abs(board[i][j]))

As for how to scan the board, we have two approaches. Let's call a line any row or column of the board.

For each line, we could use a sliding window (or itertools.groupby in Python) to find contiguous segments of the same character. If any of these segments have length 3 or more, we should flag them.

Alternatively, for each line, we could look at each width-3 slice of the line: if they are all the same, then we should flag those 3.

After, we can crush the candy by setting all flagged board cells to zero.

Gravity Step

For each column, we want all the candy to go to the bottom. One way is to iterate through and keep a stack of the (uncrushed) candy, popping and setting as we iterate through the column in reverse order.

Alternatively, we could use a sliding window approach, maintaining a read and write head. As the read head iterates through the column in reverse order, when the read head sees candy, the write head will write it down and move one place. Then, the write head will write zeroes to the remainder of the column.

We showcase the simplest approaches to these steps in the solutions below.

|  |
| --- |
| class Solution {  public int[][] candyCrush(int[][] board) {  int R = board.length, C = board[0].length;  boolean todo = false;  for (int r = 0; r < R; ++r) {  for (int c = 0; c + 2 < C; ++c) {  int v = Math.abs(board[r][c]);  if (v != 0 && v == Math.abs(board[r][c+1]) && v == Math.abs(board[r][c+2])) {  board[r][c] = board[r][c+1] = board[r][c+2] = -v;  todo = true;  }  }  }  for (int r = 0; r + 2 < R; ++r) {  for (int c = 0; c < C; ++c) {  int v = Math.abs(board[r][c]);  if (v != 0 && v == Math.abs(board[r+1][c]) && v == Math.abs(board[r+2][c])) {  board[r][c] = board[r+1][c] = board[r+2][c] = -v;  todo = true;  }  }  }  for (int c = 0; c < C; ++c) {  int wr = R - 1;  for (int r = R-1; r >= 0; --r)  if (board[r][c] > 0)  board[wr--][c] = board[r][c];  while (wr >= 0)  board[wr--][c] = 0;  }  return todo ? candyCrush(board) : board;  }  } |

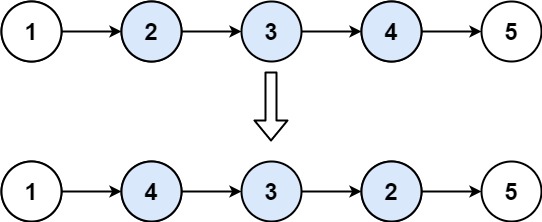
\*\*Complexity Analysis\*\*

* Time Complexity: O((R\*C)^2)*O*((*R*∗*C*)2), where R, C*R*,*C* is the number of rows and columns in board. We need O(R\*C)*O*(*R*∗*C*) to scan the board, and we might crush only 3 candies repeatedly.
* Space Complexity: O(1)*O*(1) additional complexity, as we edit the board in place.

**Reverse Linked List II**

Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return the reversed list.

**Example 1:**



**Input:** head = [1,2,3,4,5], left = 2, right = 4

**Output:** [1,4,3,2,5]

**Example 2:**

**Input:** head = [5], left = 1, right = 1

**Output:** [5]

**Constraints:**

* The number of nodes in the list is n.
* 1 <= n <= 500
* -500 <= Node.val <= 500
* 1 <= left <= right <= n

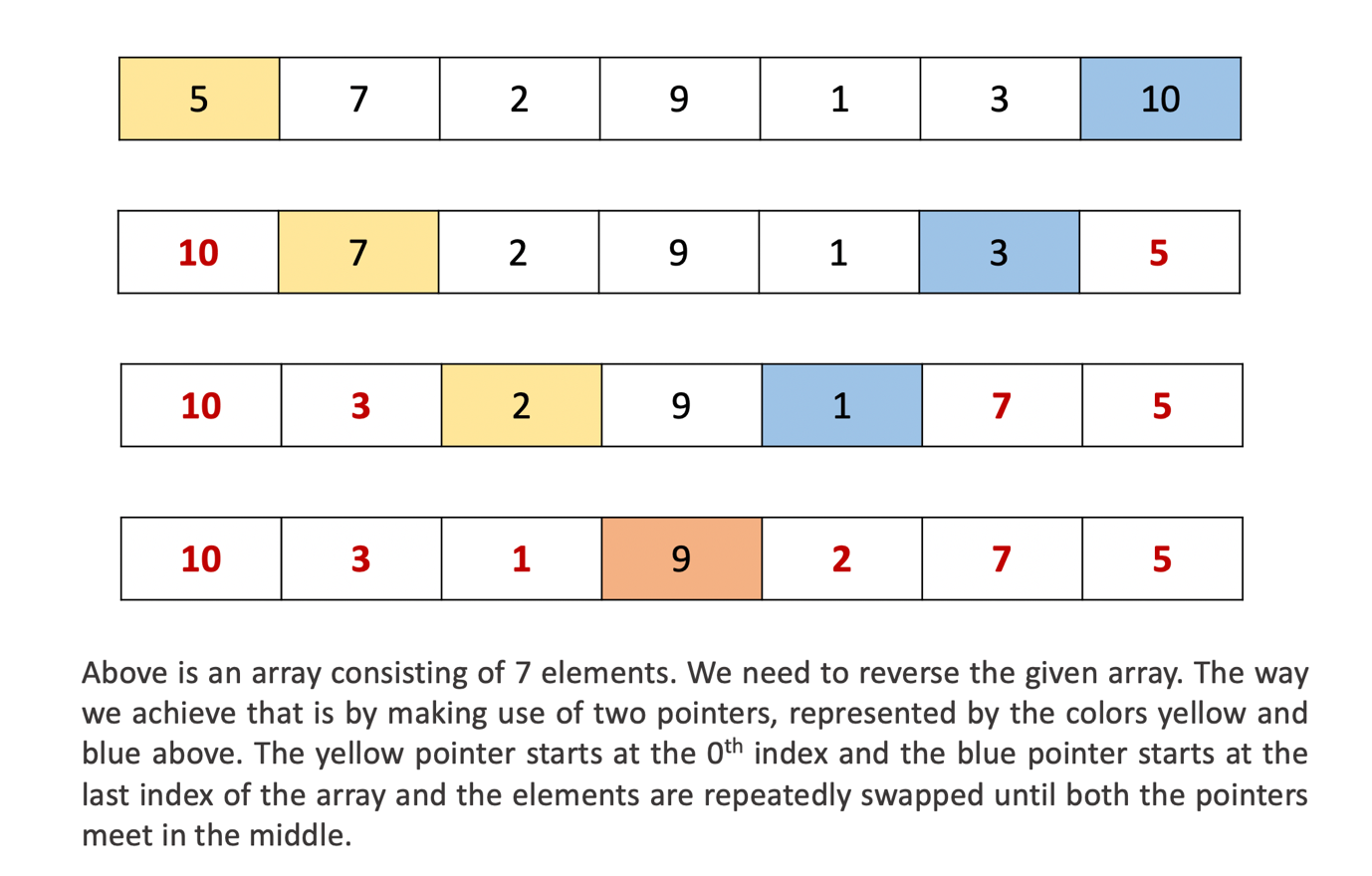
**Follow up:** Could you do it in one pass?

## Solution

#### **Approach 1: Recursion**

**Intuition**

The idea for linked list reversal using recursion springs from a similar idea that we use for reversing an array. If we want to reverse an array, the huge advantage that we have is the availability of indexes. So, what we can do there is to simply have two pointers, one at the beginning of the array and one at the end. We repeatedly swap elements pointed to by these two pointers and we move both the pointers towards the center of the array. Let's quickly look at this simple algorithm on a sample array before we move on to linked lists.



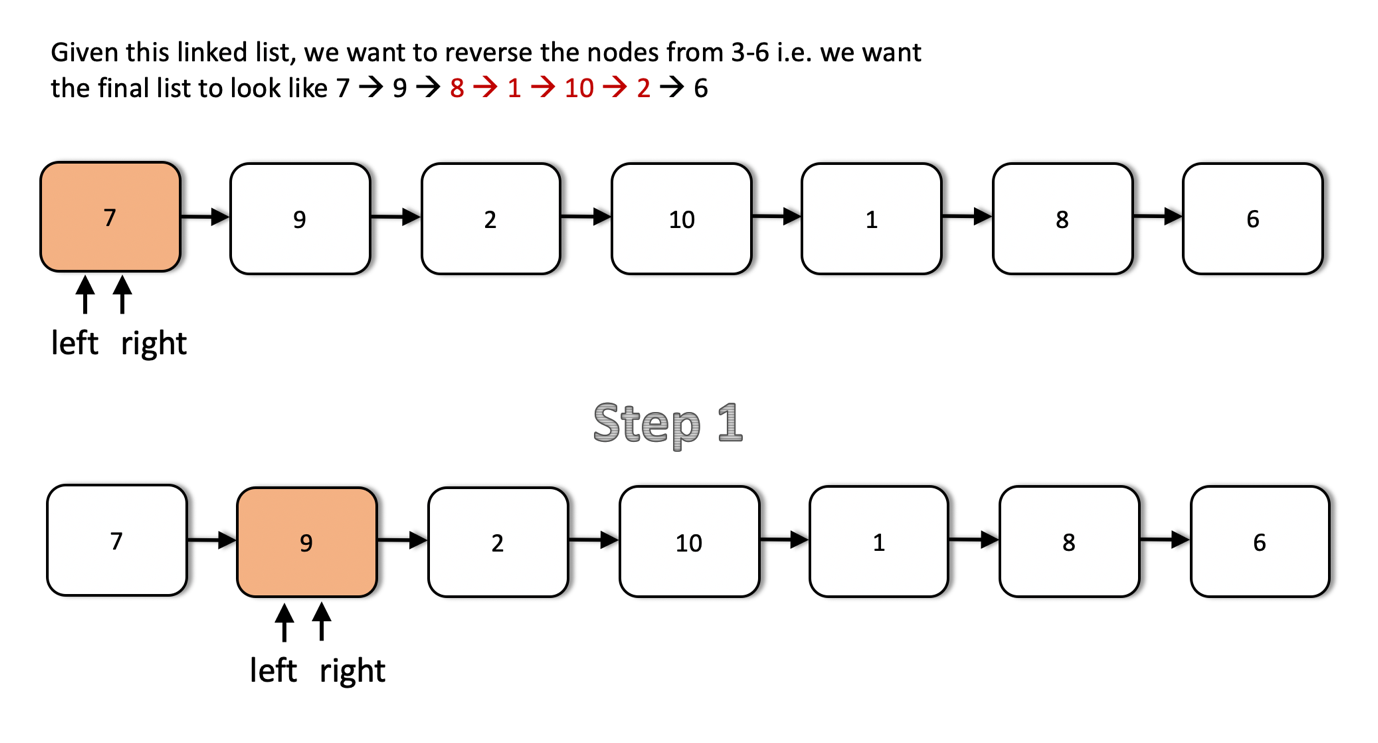
The first approach for reversing a portion of the given linked list is based on the similar idea expressed above. Essentially, we want two different pointers, one at the m^{th}*mth* node from the beginning and another one from the n^{th}*nth* node from the beginning. Once we have such pointers in place, we can repeatedly swap the data between the nodes and progress these pointers towards each other like we saw in the case of an array.

However, we don't have any backward pointers in our linked list and neither do we have any indexes. So, we rely on recursion to simulate the backward pointer. Essentially, the backtracking process in a recursion will help us in simulating the backward movement of the pointer from the n^{th}*nth* node in the linked list towards the center.

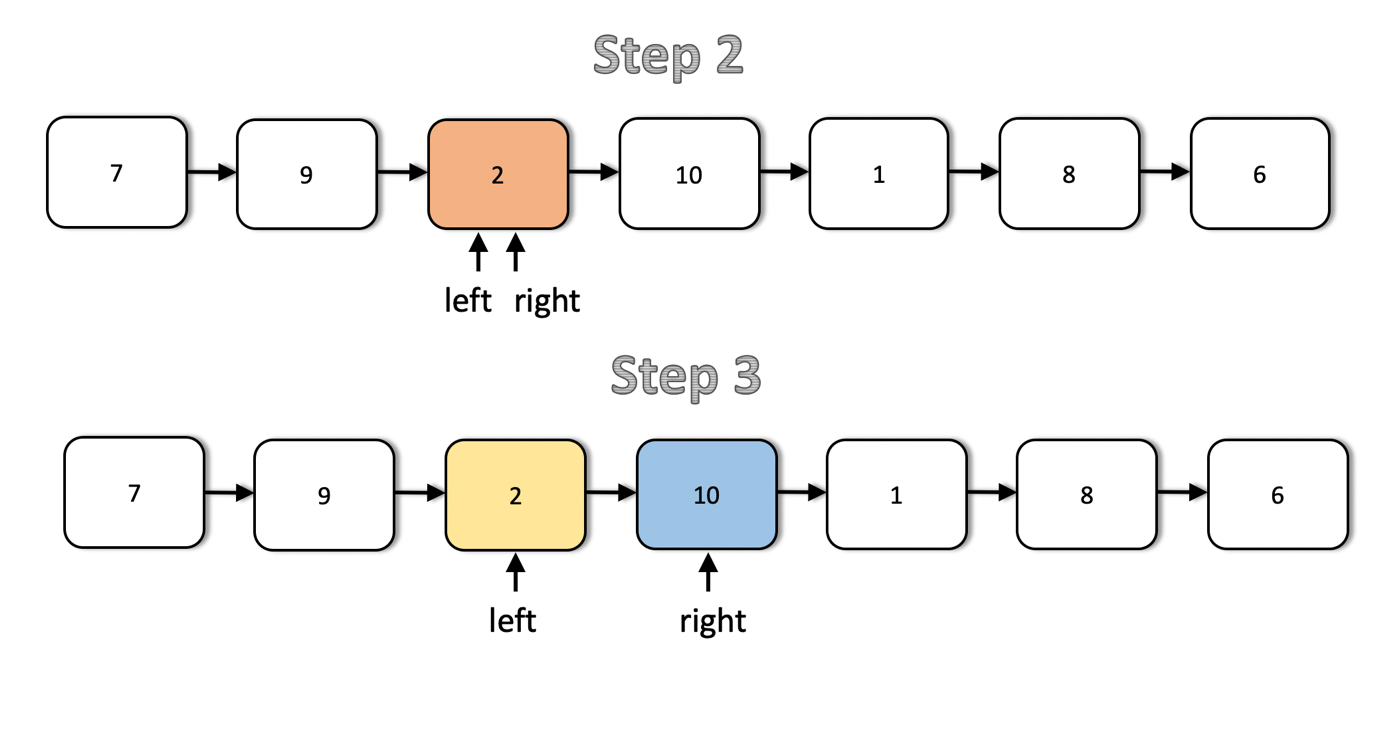
**Algorithm**

1. We define a recursion function that will do the job of reversing a portion of the linked list.
2. Let's call this function recurse. The function takes in 3 parameters: m being the starting point of the reversal, n being the ending point for the reversal, and a pointer right which will start at the n^{th}*nth* node in the linked list and move backwards with the backtracking of the recursion. If this is not clear at the moment, the diagrams that follow will help.
3. Additionally, we have a pointer called left which starts from the m^{th}*mth* node in the linked list and moves forward. In Python, we have to take a global variable for this which get's changed with recursion. In other languages, where changes made in function calls persist, we can consider this pointer as an additional variable for the function recurse.
4. In a recursion call, given m, n, and right, we check if n == 1. If this is the case, we don't need to go any further.
5. Until we reach n = 1, we keep moving the right pointer one step forward and after doing that, we make a recursive call with the value of n decreased by 1. At the same time, we keep on moving the left pointer forward until m == 1. When we refer to a pointer being moved forward, it essentially means pointer.next.
6. So we backtrack as soon as n reaches 1. At that point of time, the right pointer is at the last node of the sublist we want to reverse and the left has already reached the first node of this sublist. So, we swap out the data and move the left pointer one step forward using left = left.next. We need this change to persist across the backtracking process.
7. From there on, every time we backtrack, the right pointer moves one step backwards. This is the simulation we've been mentioning all along. The backward movement is simulated by backtracking.
8. We stop the swaps when either right == left, which happens if the sublist size is odd, or, right.next == left which happens when during the backtracking process for an even sized sublist, the right pointer crosses left. We use a global boolean flag for stopping the swaps once these conditions are met.

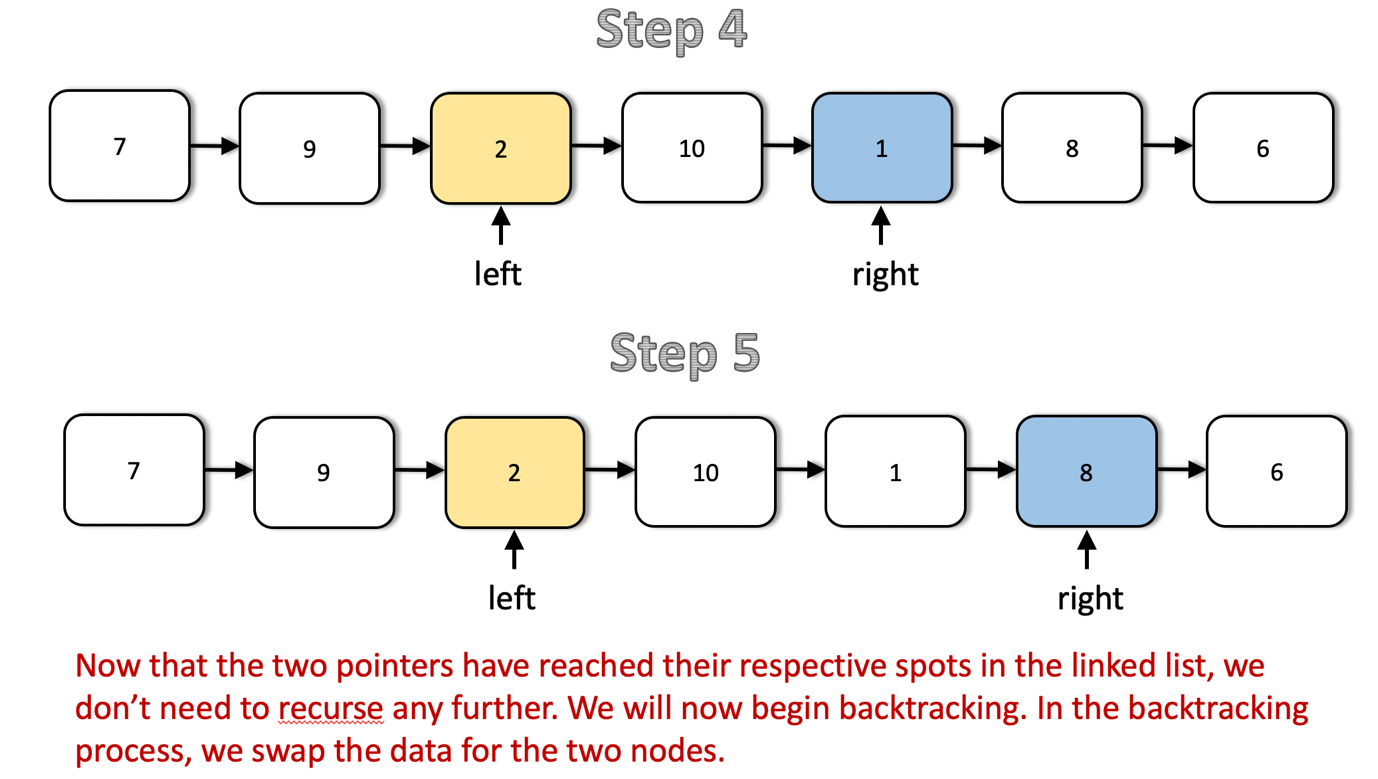
Let's look at a series of diagrams explaining the process on a sample linked list. Hopefully, things would be clearer after this.



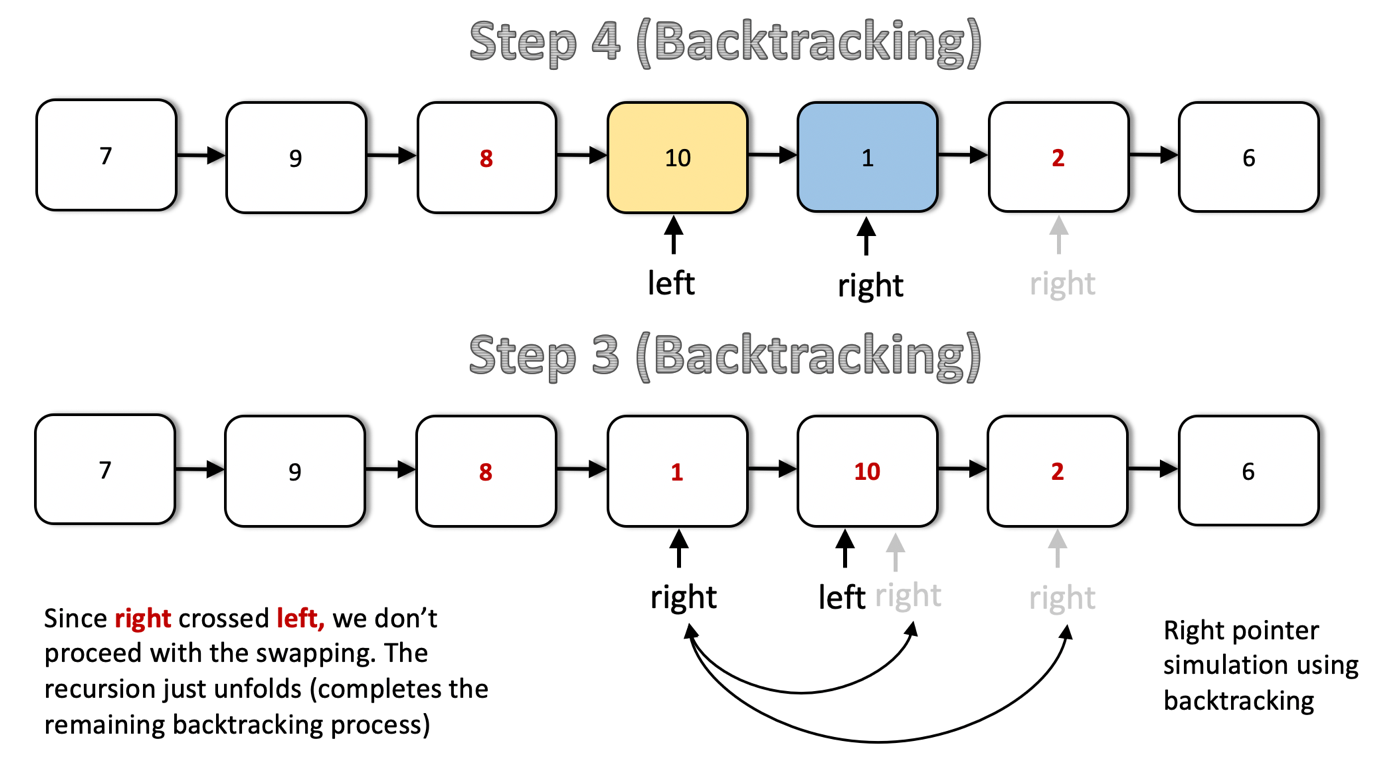
This is the first step in the recursion process. We have a list given to us and the left and the right pointers start off from the head of the linked list. The first step makes a recursive call with updated values of m and n i.e. their values each reduced by 1. Also, the left and the right pointers move one step forward in the linked list.



The next two steps show the movement of the left and the right pointers in the list. Notice that after the second step, the left pointer reaches it's designated spot. So, we don't move it any further. Only the right pointer progresses from here on out until it reaches node 6.



As we can see, after the step 5, both the pointers are in their designated spots in the list and we can start the backtracking process. We don't recurse further. The operation performed during the backtracking is swapping of data between the left and right nodes.



The right pointer crosses the left pointer after step 3 (backtracking) as can be seen above and by that point, we have already reversed the required portion of the linked list. We needed the output list to be [7 → 9 → 8 → 1 → 10 → 2 → 6] and that's what we have. So, we don't perform any more swaps and in the code, we can use a global boolean flag to stop the swapping after a point. We can't really break out of recursion per say.

|  |
| --- |
| class Solution {  // Object level variables since we need the changes  // to persist across recursive calls and Java is pass by value.  private boolean stop;  private ListNode left;  public void recurseAndReverse(ListNode right, int m, int n) {  // base case. Don't proceed any further  if (n == 1) {  return;  }  // Keep moving the right pointer one step forward until (n == 1)  right = right.next;  // Keep moving left pointer to the right until we reach the proper node  // from where the reversal is to start.  if (m > 1) {  this.left = this.left.next;  }  // Recurse with m and n reduced.  this.recurseAndReverse(right, m - 1, n - 1);  // In case both the pointers cross each other or become equal, we  // stop i.e. don't swap data any further. We are done reversing at this  // point.  if (this.left == right || right.next == this.left) {  this.stop = true;  }  // Until the boolean stop is false, swap data between the two pointers  if (!this.stop) {  int t = this.left.val;  this.left.val = right.val;  right.val = t;  // Move left one step to the right.  // The right pointer moves one step back via backtracking.  this.left = this.left.next;  }  }  public ListNode reverseBetween(ListNode head, int m, int n) {  this.left = head;  this.stop = false;  this.recurseAndReverse(head, m, n);  return head;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*) since we process all the nodes at-most twice. Once during the normal recursion process and once during the backtracking process. During the backtracking process we only just swap half of the list if you think about it, but the overall complexity is O(N)*O*(*N*).
* Space Complexity: O(N)*O*(*N*) in the worst case when we have to reverse the entire list. This is the space occupied by the recursion stack.

#### **Approach 2: Iterative Link Reversal.**

**Intuition**

In the previous approach, we looked at an algorithm for reversing a portion of the given linked list such that the underlying structure doesn't change. We only modified the values of the nodes for achieving the reversal. However, it may so happen that you cannot change the data available in the nodes. In that scenario, we have to modify the links themselves to achieve the reversal.

Essentially, starting from the node at position m and all the way up to n, we reverse the next pointers for all the nodes in between. Let's look at the algorithm for achieving this.

**Algorithm**

Before looking at the algorithm, it's important to understand how the link reversal will work and what set of pointers will be required for the same. Let's say we have a linked list consisting of three different nodes, A → B → C and we want to reverse the links between the nodes and obtain A ← B ← C.

Suppose we have at our disposal, two pointers. One of them points to the node A and the other one points to the node B. Let's call these pointers prev and cur respectively. We can simply use these two pointers to reverse the link between A and B.

cur.next = prev

The only problem with this is, we don't have a way of progressing further i.e. once we do this, we can't reach the node C. That's why we need a third pointer that will help us continue the link reversal process. So, we do the following instead.

third = cur.next

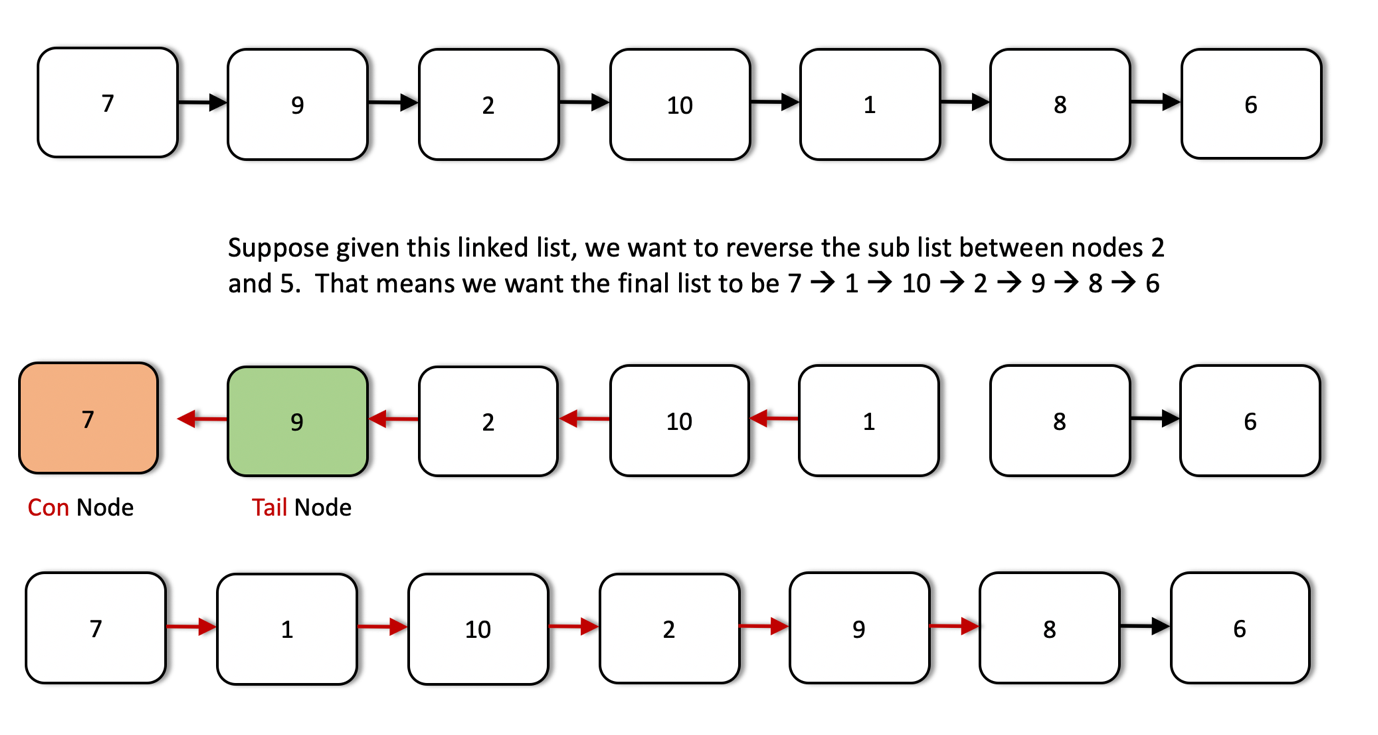
cur.next = prev

prev = cur

cur = third

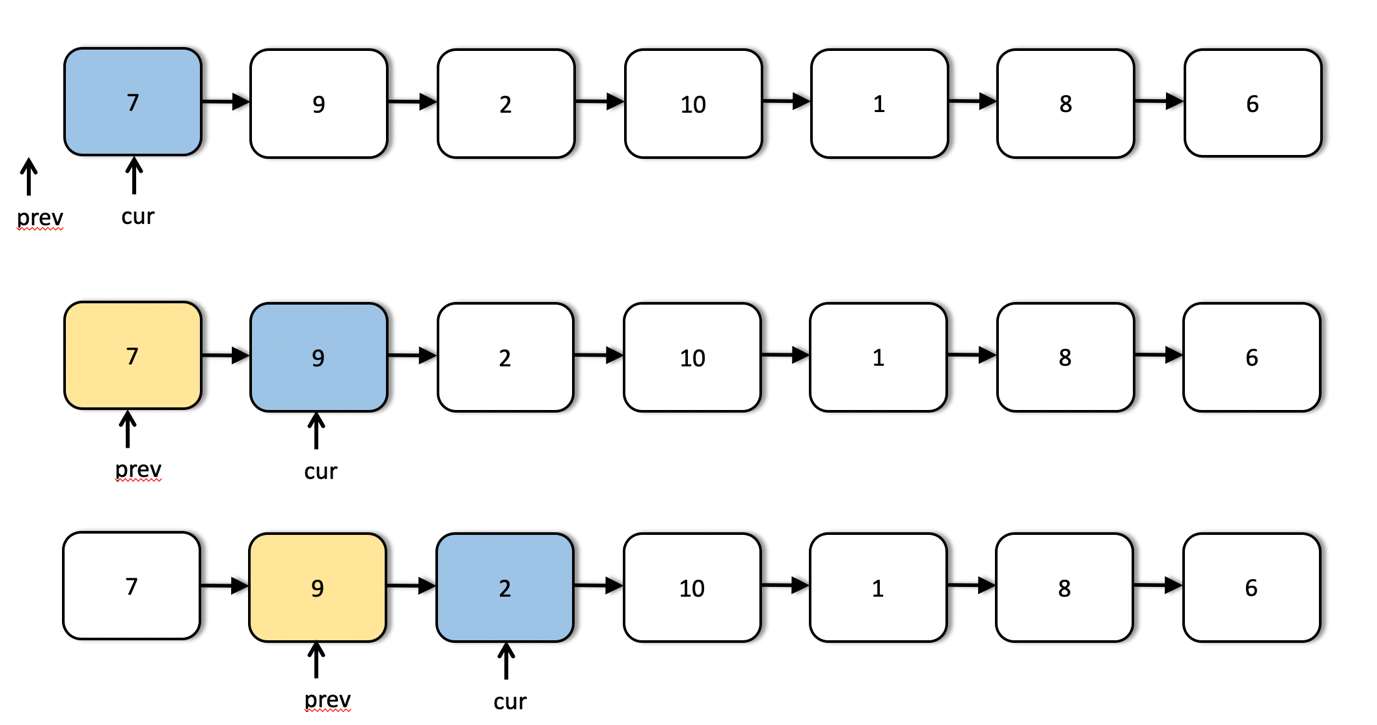
We do the above iteratively and we will achieve what the question asks us to do. Let's look at the steps for the algorithm now.

1. We need two pointers, prev and cur as explained above.
2. The prev pointer should be initialized to None initially while cur is initialized to the head of the linked list.
3. We progress the cur pointer one step at a time and the prev pointer follows it.
4. We keep progressing the two pointers in this way until the cur pointer reaches the m^{th}*mth* node from the beginning of the list. This is the point from where we start reversing our linked list.
5. An important thing to note here is the usage of two additional pointers which we will call as tail and con. The tail pointer points to the m^{th}*mth* node from the beginning of the linked list and we call it a tail pointer since this node becomes the tail of the reverse sublist. The con points to the node one before m^{th}*mth* node and this connects to the new head of the reversed sublist. Let's take a look at a figure to understand these two pointers better.

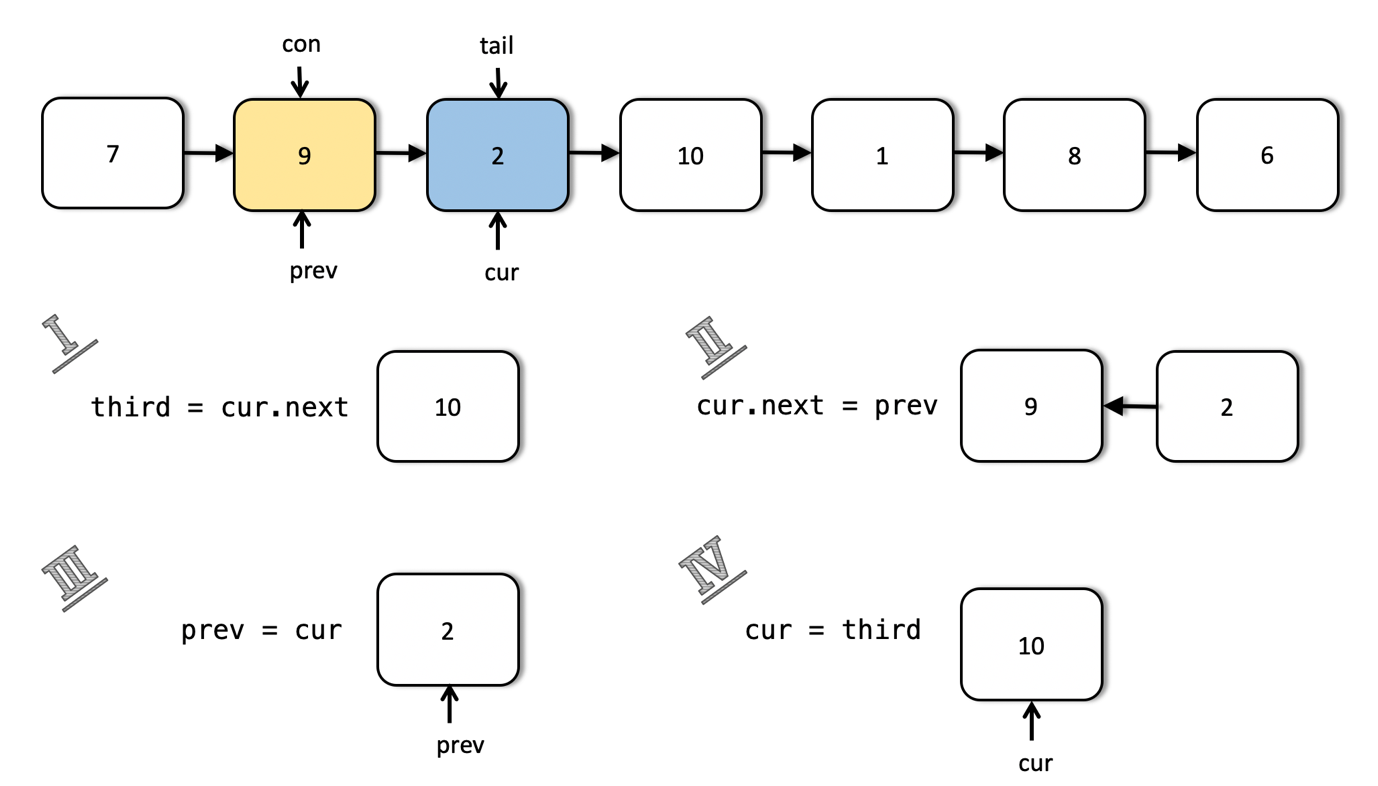


1. The tail and the con pointers are set once initially and then used in the end to finish the linked list reversal.
2. Once we reach the m^{th}*mth* node, we iteratively reverse the links as explained before using the two pointers. We keep on doing this until we are done reversing the link (next pointer) for the n^{th}*nth* node. At that point, the prev pointer would point to the n^{th}*nth* node.
3. We use the con pointer to attach to the prev pointer since the node now pointed to by the prev pointer (the n^{th}*nth* node from the beginning) will come in place of the m^{th}*mth* node due after the reversal. Similarly, we will make use of the tail pointer to connect to the node next to the prev node i.e. (n+1)^{th}(*n*+1)*th* node from the beginning.

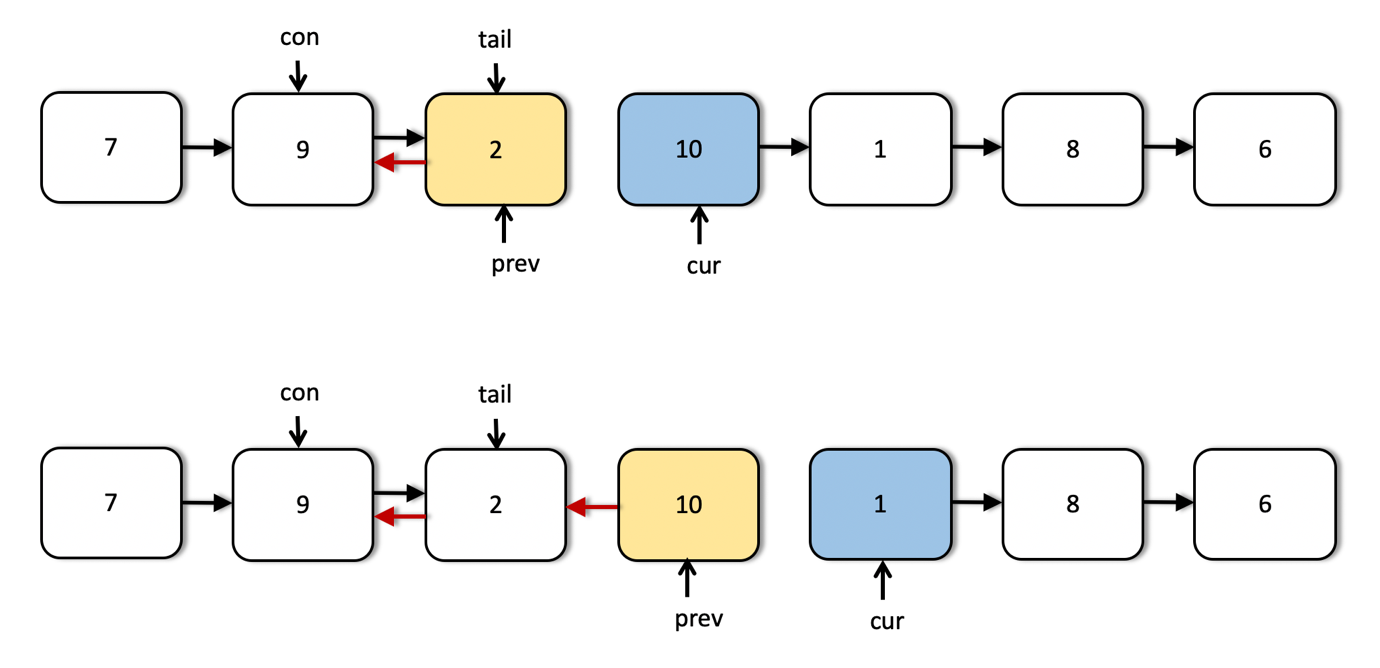
Let's have a look at the algorithm execute on a sample linked list to make the use case for all these pointers clearer. We are given a linked list initially with elements 7 → 9 → 2 → 10 → 1 → 8 → 6 and we need to reverse the list from node 3 through 6.

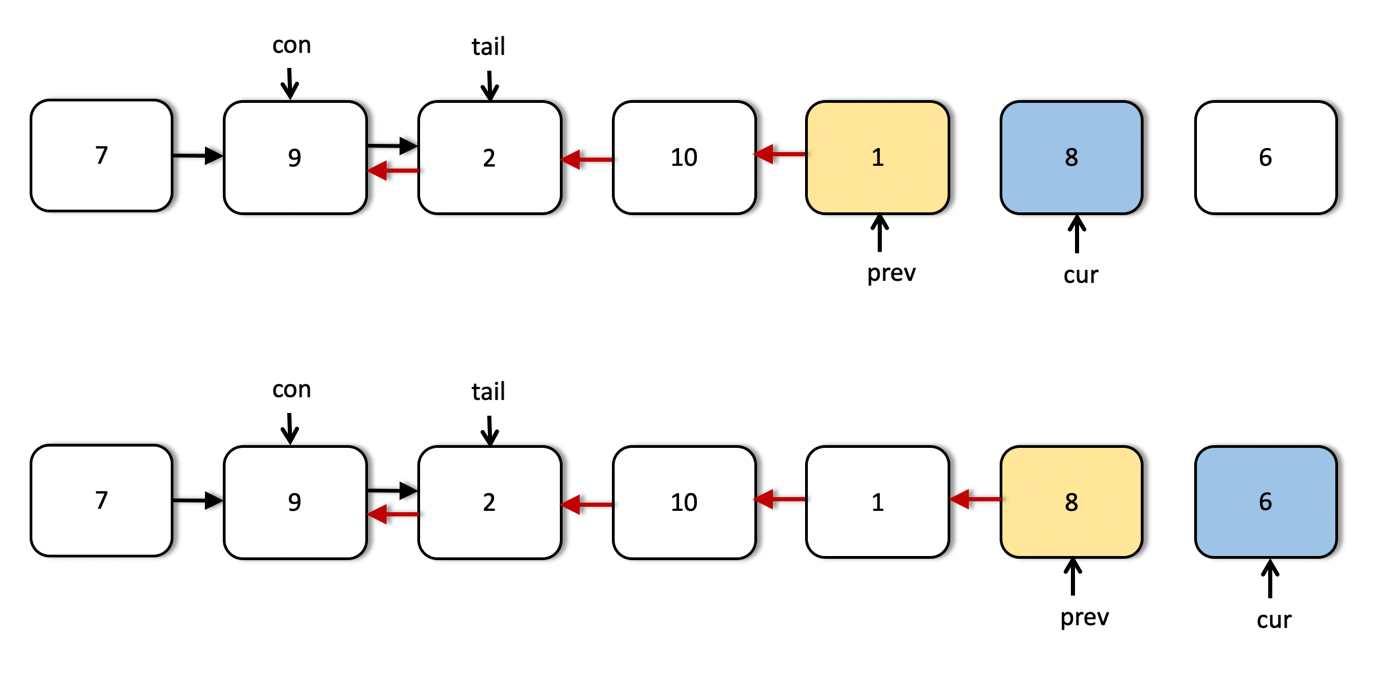


We can see the first few steps of our iterative solution above. The first step shows the initialization of the two pointers and the third step shows us the starting point for the list reversal process.

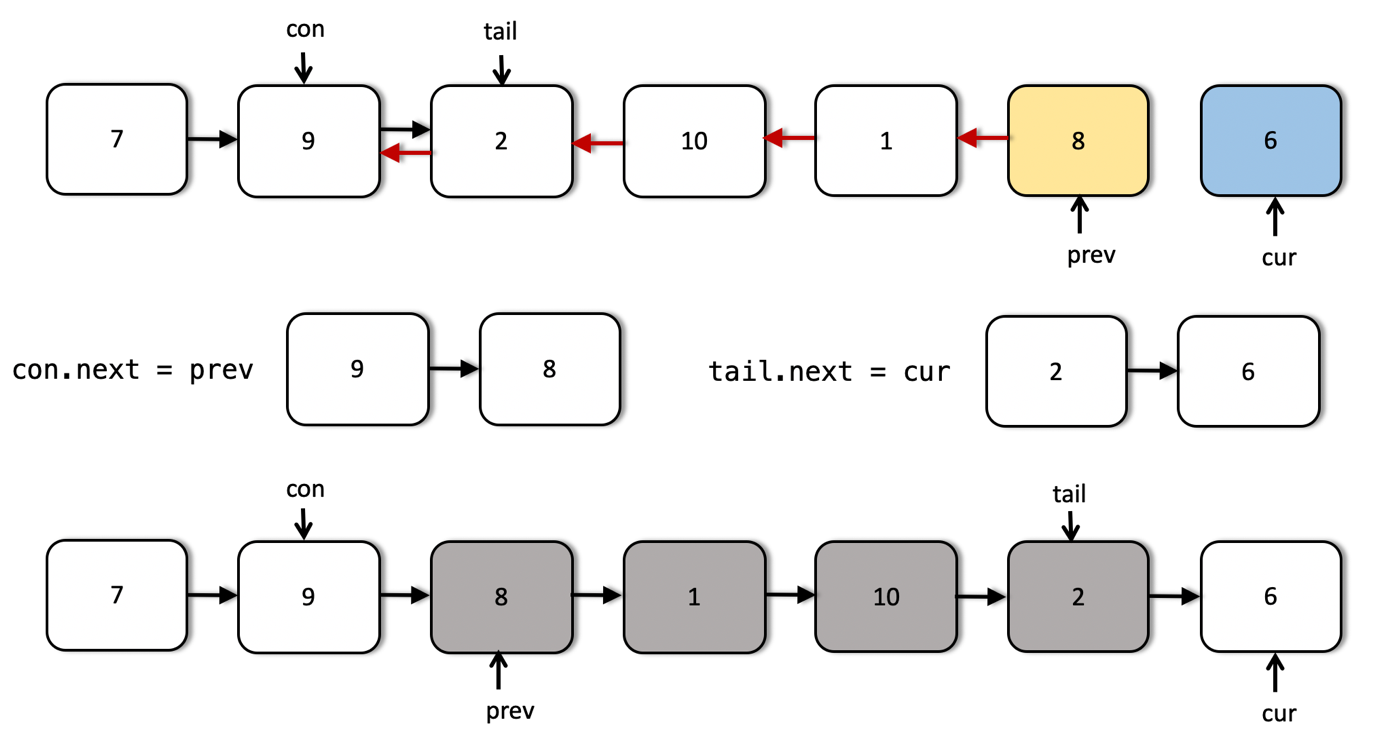


This shows us in detail how the links are reversed and how we move forward after reversing the links between two nodes. This step is done multiple times as shown in the following images.





As we can see from the above images, now the two pointers have reached their final positions. We are done reversing the sublist that we were required to do i.e. nodes 3 through 6. However, we still have to fix some connections. The next image explains how we use the tail and con pointers to make the final connections.



|  |
| --- |
| class Solution {  public ListNode reverseBetween(ListNode head, int m, int n) {  // Empty list  if (head == null) {  return null;  }  // Move the two pointers until they reach the proper starting point  // in the list.  ListNode cur = head, prev = null;  while (m > 1) {  prev = cur;  cur = cur.next;  m--;  n--;  }  // The two pointers that will fix the final connections.  ListNode con = prev, tail = cur;  // Iteratively reverse the nodes until n becomes 0.  ListNode third = null;  while (n > 0) {  third = cur.next;  cur.next = prev;  prev = cur;  cur = third;  n--;  }  // Adjust the final connections as explained in the algorithm  if (con != null) {  con.next = prev;  } else {  head = prev;  }  tail.next = cur;  return head;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*) considering the list consists of N*N* nodes. We process each of the nodes at most once (we don't process the nodes after the n^{th}*nth* node from the beginning.
* Space Complexity: O(1)*O*(1) since we simply adjust some pointers in the original linked list and only use O(1)*O*(1) additional memory for achieving the final result.

**Sort Characters By Frequency**

Given a string, sort it in decreasing order based on the frequency of characters.

**Example 1:**

**Input:**

"tree"

**Output:**

"eert"

**Explanation:**

'e' appears twice while 'r' and 't' both appear once.

So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.

**Example 2:**

**Input:**

"cccaaa"

**Output:**

"cccaaa"

**Explanation:**

Both 'c' and 'a' appear three times, so "aaaccc" is also a valid answer.

Note that "cacaca" is incorrect, as the same characters must be together.

**Example 3:**

**Input:**

"Aabb"

**Output:**

"bbAa"

**Explanation:**

"bbaA" is also a valid answer, but "Aabb" is incorrect.

Note that 'A' and 'a' are treated as two different characters.

## Solution

#### **Remember, Strings are Immutable!**

The input type for this question is a String. When dealing with Strings, we need to be careful to not inadvertently convert what should have been an O(n)*O*(*n*) algorithm into an O(n^2)*O*(*n*2) one.

Strings in most programming languages are **immutable**. This means that once a String is created, we cannot modify it. We can only create a new String. Consider the following Java code.

String a = "Hello ";

a += "Leetcode";

This code creates a String called a with the value "Hello ". It then sets a to be a new String, made with the letters from the old a and the additional letters "Leetcode". It then assigns this new String to the variable a, throwing away the reference to the old one. It does NOT actually "modify" a.

For the most part, we don't run into problems with Strings being treated like this. But consider this code for reversing a String.

String s = "Hello There";

String reversedString = "";

for (int i = s.length() - 1; i >= 0; i--) {

reversedString += s.charAt(i);

}

System.out.println(reversedString);

Each time a character is added to reverseString, a new String is created. Creating a new String has a cost of n*n*, where n*n* is the length of the String. The result? Simply reversing a String has a cost of O(n^2)*O*(*n*2) using the above algorithm.

The solution is to use a StringBuilder. A StringBuilder collects up the characters that will be converted into a String so that only one String needs to be created—once all the characters are ready to go. Recall that inserting an item at the end of an Array has a cost of O(1)*O*(1), and so the total cost of inserting the n*n* characters into the StringBuilder is O(n)*O*(*n*), and it is also O(n)*O*(*n*) to then convert that StringBuilder into a String, giving a total of O(n)*O*(*n*).

String s = "Hello There";

StringBuilder sb = new StringBuilder();

for (int i = s.length() - 1; i >= 0; i--) {

sb.append(s.charAt(i));

}

String reversedString = sb.toString();

System.out.println(reversedString);

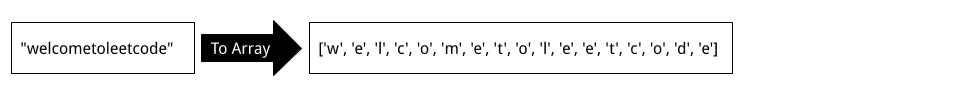
If you're unsure what to do for your particular programming language, it shouldn't be too difficult to find using a web search. The algorithms provided in the solutions here all do string building efficiently.

#### **Approach 1: Arrays and Sorting**

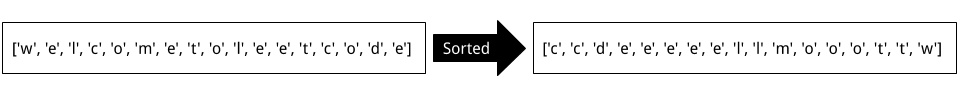
**Intuition**

In order to sort the characters by frequency, we firstly need to know how many of each there are. One way to do this is to sort the characters by their numbers so that identical characters are side-by-side (all characters in a programming language are identified by a unique number). Then, knowing how many times each appears will be a lot easier.

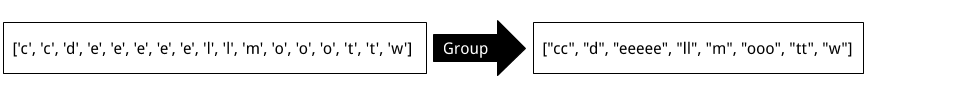
Because Strings are **immutable** though, we cannot sort the String directly. Therefore, we'll need to start by converting it from a String to an Array of characters.



Now that we have an Array, we can sort it, which will make all identical characters side-by-side.

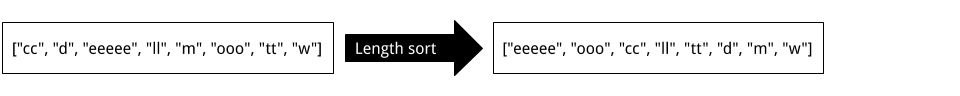


There are a few different ways we can go from here. One easy-to-understand way is to create a new Array of Strings. Each String in the list will consist of one of the unique characters from the sorted characters Array.

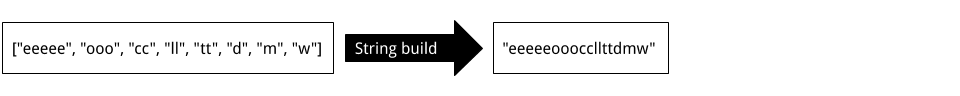


Remember: do this process using StringBuilders, not naïve String appending! (See the first section of this article if you're confused).

The next step is to sort this Array of Strings by length. To do this, we'll need to implement a suitable **Comparator**. Recall that there is no requirement for characters of the same frequency to appear in a specific order.



Finally, we can convert this Array of Strings into a single String. In Java, this can be done by passing the Array into a StringBuilder and then calling .toString(...) on it.



**Algorithm**

|  |
| --- |
| public String frequencySort(String s) {    if (s == null || s.isEmpty()) return s;    // Create a sorted Array of chars.  char[] chars = s.toCharArray();  Arrays.sort(chars);    // Convert identical chars into single Strings.  List<String> charStrings = new ArrayList<String>();  StringBuilder currentString = new StringBuilder();  currentString.append(chars[0]);  for (int i = 1; i < chars.length; i++) {  if (chars[i] != chars[i - 1]) {  charStrings.add(currentString.toString());  currentString = new StringBuilder();  }  currentString.append(chars[i]);  }  charStrings.add(currentString.toString());    // Our comparator is (a, b) -> b.length() - a.length().  // If a is longer than b, then a negative number will be returned  // telling the sort algorithm to place a first. Otherwise, a positive  // number will be returned, telling it to place a second.  // This results in a longest to shortest sorted list of the strings.  Collections.sort(charStrings, (a, b) -> b.length() - a.length());    // Use StringBuilder to build the String to return.  StringBuilder sb = new StringBuilder();  for (String str : charStrings) sb.append(str);  return sb.toString();  } |

**Complexity Analysis**

Let n*n* be the length of the input String.

* Time Complexity : O(n \, \log \, n)*O*(*n*log*n*).

The first part of the algorithm, converting the String to a List of characters, has a cost of O(n)*O*(*n*), because we are adding n*n* characters to the end of a List.

The second part of the algorithm, sorting the List of characters, has a cost of O(n \, \log \, n)*O*(*n*log*n*).

The third part of the algorithm, grouping the characters into Strings of similar characters, has a cost of O(n)*O*(*n*) because each character is being inserted once into a StringBuilder and once converted into a String.

Finally, the fourth part of the algorithm, sorting the Strings by length, has a worst case cost of O(n)*O*(*n*), which occurs when all the characters in the input String are unique.

Because we drop constants and insignificant terms, we get O(n \, \log \, n) + 3 \cdot O(n) = O(n \, \log \, n)*O*(*n*log*n*)+3⋅*O*(*n*)=*O*(*n*log*n*).

Be careful with your own implementation—if you didn't do the string building process in a sensible way, then your solution could potentially be O(n^2)*O*(*n*2).

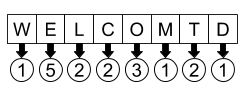
* Space Complexity : O(n)*O*(*n*).

It is impossible to do better with the space complexity, because Strings are immutable. The List of characters, List of Strings, and the final output String, are all of length n*n*, so we have a space complexity of O(n)*O*(*n*).

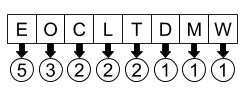
#### **Approach 2: HashMap and Sort**

**Intuition**

Another approach is to use a HashMap to count how many times each character occurs in the String; the keys are characters and the values are frequencies.



Next, extract a copy of the keys from the HashMap and sort them by frequency using a Comparator that looks at the HashMap values to make its decisions.



Finally, initialise a new StringBuilder and then iterate over the list of sorted characters (sorted by frequency). Look up the values in the HashMap to know how many of each character to append to the StringBuilder.

**Algorithm**

|  |
| --- |
| public String frequencySort(String s) {  // Count up the occurances.  Map<Character, Integer> counts = new HashMap<>();  for (char c : s.toCharArray()) {  counts.put(c, counts.getOrDefault(c, 0) + 1);  }    // Make a list of the keys, sorted by frequency.  List<Character> characters = new ArrayList<>(counts.keySet());  Collections.sort(characters, (a, b) -> counts.get(b) - counts.get(a));  // Convert the counts into a string with a sb.  StringBuilder sb = new StringBuilder();  for (char c : characters) {  int copies = counts.get(c);  for (int i = 0; i < copies; i++) {  sb.append(c);  }  }  return sb.toString();  } |

**Complexity Analysis**

Let n*n* be the length of the input String and k*k* be the number of unique characters in the String.

We know that k ≤ n*k*≤*n*, because there can't be more unique characters than there are characters in the String. We also know that k*k* is somewhat bounded by the fact that there's only a finite number of characters in Unicode (or ASCII, which I suspect is all we need to worry about for this question).

There are two ways of approaching the complexity analysis for this question.

1. We can disregard k*k*, and consider that in the worst case, *k = n*.
2. We can consider k*k*, recognising that the number of unique characters is not infinite. This is more accurate for real world purposes.

I've provided analysis for both ways of approaching it. I choose not to bring it up for the previous approach, because it made no difference there.

* Time Complexity : O(n \, \log \, n)*O*(*n*log*n*) OR O(n + k \, \log \, k)*O*(*n*+*k*log*k*).

Putting the characterss into the HashMap has a cost of O(n)*O*(*n*), because each of the n*n* characterss must be put in, and putting each in is an O(1)*O*(1) operation.

Sorting the HashMap keys has a cost of O(k \, \log \, k)*O*(*k*log*k*), because there are k*k* keys, and this is the standard cost for sorting. If only using n*n*, then it's O(n \, \log \, n)*O*(*n*log*n*). For the previous question, the sort was carried out on n*n* items, not k*k*, so was possibly a lot worse.

Traversing over the sorted keys and building the String has a cost of O(n)*O*(*n*), as n*n* characters must be inserted.

Therefore, if we're only considering n*n*, then the final cost is O(n \, \log \, n)*O*(*n*log*n*).

Considering k*k* as well gives us O(n + k \, \log \, k)*O*(*n*+*k*log*k*), because we don't know which is largest out of n*n* and k \, \log \, k*k*log*k*. We do, however, know that in total this is less than or equal to O(n \, \log \, n)*O*(*n*log*n*).

* Space Complexity : O(n)*O*(*n*).

The HashMap uses O(k)*O*(*k*) space.

However, the StringBuilder at the end dominates the space complexity, pushing it up to O(n)*O*(*n*), as every character from the input String must go into it. Like was said above, it's impossible to do better with the space complexity here.

What's interesting here is that if we only consider n*n*, the time complexity is the same as the previous approach. But by considering k*k*, we can see that the difference is potentially substantial.

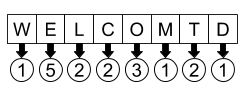
#### **Approach 3: Multiset and Bucket Sort**

**Intuition**

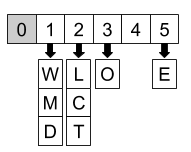
While the second approach is probably adequate for an interview, there is actually a way of solving this problem with a time complexity of O(n)*O*(*n*).

Firstly, observe that because all of the characters came out of a String of length n*n*, the maximum frequency for any one character is n*n*. This means that once we've determined all the letter frequencies using a HashMap, we can sort them in O(n)*O*(*n*) time using **Bucket Sort**. Recall that for our previous approaches, we used comparison-based sorts, which have a cost of O(n \, \log \, n)*O*(*n*log*n*).

This was the HashMap from earlier.



Recall that **Bucket Sort** is the sorting algorithm where items are placed at Array indexes based on their values (the indexes are called "buckets"). For this problem, we'll need to have a List of characters at each index. For example, here is how our String from before goes into the buckets.



While we could simply make our bucket Array length n*n*, we're best to just look for the maximum value (frequency) in the HashMap. That way, we only use as much space as we need, and won't need to iterate over heaps of empty buckets during the next phase.

Finally, we need to iterate over the buckets, starting with the largest and ending with the smallest, building up the string in much the same way as we did before.

**Algorithm**

|  |
| --- |
| public String frequencySort(String s) {    if (s == null || s.isEmpty()) return s;    // Count up the occurances.  Map<Character, Integer> counts = new HashMap<>();  for (char c : s.toCharArray()) {  counts.put(c, counts.getOrDefault(c, 0) + 1);  }    int maximumFrequency = Collections.max(counts.values());    // Make the list of buckets and apply bucket sort.  List<List<Character>> buckets = new ArrayList<>();  for (int i = 0; i <= maximumFrequency; i++) {  buckets.add(new ArrayList<Character>());  }  for (Character key : counts.keySet()) {  int freq = counts.get(key);  buckets.get(freq).add(key);  }  // Build up the string.  StringBuilder sb = new StringBuilder();  for (int i = buckets.size() - 1; i >= 1; i--) {  for (Character c : buckets.get(i)) {  for (int j = 0; j < i; j++) {  sb.append(c);  }  }  }  return sb.toString();  } |

**Complexity Analysis**

Let n*n* be the length of the input String. The k*k* (number of unique characters in the input String that we considered for the last approach makes no difference this time).

* Time Complexity : O(n)*O*(*n*).

Like before, the HashMap building has a cost of O(n)*O*(*n*).

The bucket sorting is O(n)*O*(*n*), because inserting items has a cost of O(k)*O*(*k*) (each entry from the HashMap), and building the buckets initially has a worst case of O(n)*O*(*n*) (which occurs when k = 1*k*=1). Because k ≤ n*k*≤*n*, we're left with O(n)*O*(*n*).

So in total, we have O(n)*O*(*n*).

It'd be impossible to do better than this, because we need to look at each of the n*n* characters in the input String at least once.

* Space Complexity : O(n)*O*(*n*).

Same as above. The bucket Array also uses O(n)*O*(*n*) space, because its length is at most n*n*, and there are k*k* items across all the buckets.