**Amazon**

**Introduction**

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

We compiled this list thoroughly so you can save time and get well-prepared for an Amazon interview.

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

Arrays and Strings

 Two Sum. - Array

 Longest Substring Without Repeating Characters - HashTable

 String to Integer (atoi) - EZ

 Container With Most Water - Hard

**Integer to Roman**

 Roman to Integer - EZ

 3Sum - Medium

**3Sum Closest**

 Implement strStr() - EZ

 Rotate Image - EZ

 Group Anagrams - Medium

 Minimum Window Substring - Hard

**Compare Version Numbers**

 Product of Array Except Self - Hard

 Missing Number - EZ

 Integer to English Words - FB

 First Unique Character in a String

 Valid Parentheses - EZ

**Most Common Word**

**Reorder Log Files**

 Trapping Rain Water - Hard

Linked Lists

 Add Two Numbers - Medium

 Merge Two Sorted Lists - EZ

**Reverse Nodes in k-Group**

 Copy List with Random Pointer – Hard/G/F/M

 Reverse Linked List – EZ/M

 Merge k Sorted Lists - Hard

Trees and Graphs

 Validate Binary Search Tree - EZ

 Symmetric Tree - EZ

 Binary Tree Level Order Traversal - EZ

 Binary Tree Zigzag Level Order Traversal - MS

 Binary Tree Maximum Path Sum - Hard

**Word Ladder II**

 Word Ladder - Hard

 Number of Islands - Medium

 Course Schedule - Hard

 Lowest Common Ancestor of a Binary Tree - Hard

 Diameter of Binary Tree - G

**Cut Off Trees for Golf Event**

 Flood Fill – Q&S

Recursion

 Letter Combinations of a Phone Number – Medium/G/FB/MS

 Generate Parentheses - Medium

 Word Search - Medium

 Word Search II - Hard

Sorting and Searching

 Median of Two Sorted Arrays - Hard

 Search in Rotated Sorted Array – Medium/FB

 Merge Intervals - Medium

 Two Sum II - Input array is sorted - Array

 Kth Largest Element in an Array - Medium

 Meeting Rooms II - Medium

 Top K Frequent Elements - Medium

 K Closest Points to Origin - Google

Dynamic Programming

 Longest Palindromic Substring - Medium

 Maximum Subarray – Medium/G/FB/MS/A/…

 Best Time to Buy and Sell Stock - EZ

 Word Break - Hard

 Coin Change - Medium

Design

 LRU Cache - Hard

 Min Stack – EZ QS

 Find Median from Data Stream - hard

 Serialize and Deserialize Binary Tree – Medium/G/FB/MS/AMZ

 Design Tic-Tac-Toe - Medium

 Design Search Autocomplete System - Tries

**Maximum Frequency Stack**

Others

 Reverse Integer – EZ/Google

 Second Highest Salary - SQL

**Partition Labels**

**Prison Cells After N Days**

## Arrays and Strings

Amazon likes to ask simple, basic array questions. We highly recommend you to practice First Unique Character in a String, which is a popular question being asked. We also recommend Integer to English Words.

**Integer to Roman**

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given an integer, convert it to a roman numeral.

**Example 1:**

**Input:** num = 3

**Output:** "III"

**Example 2:**

**Input:** num = 4

**Output:** "IV"

**Example 3:**

**Input:** num = 9

**Output:** "IX"

**Example 4:**

**Input:** num = 58

**Output:** "LVIII"

**Explanation:** L = 50, V = 5, III = 3.

**Example 5:**

**Input:** num = 1994

**Output:** "MCMXCIV"

**Explanation:** M = 1000, CM = 900, XC = 90 and IV = 4.

**Constraints:**

* 1 <= num <= 3999

## Solution

#### **Overview**

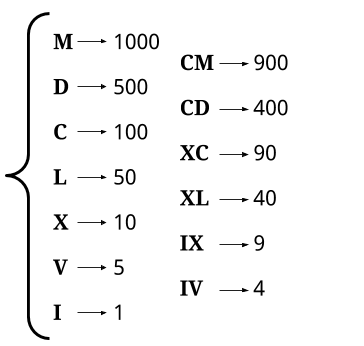
In a lot of countries, Roman Numerals are taught in elementary school-level math. This has made them a somewhat popular "easy" interview question. Unfortunately though, this ignores the fact that not everybody learned them in school, and therefore a big advantage has been given to those who did. I suspect it's also difficult for a lot of us who have learned them previously to fully appreciate how much easier prior experience makes this question. While this is very unfair, and possibly very frustrating, keep in mind that the best thing you can do is work through this question and the related question [Roman to Integer](https://leetcode.com/problems/roman-to-integer/) so that you don't get caught out by it in a real interview. In short, if you're here reading this, you've saved yourself from getting caught out by it! Thankfully, questions that rely on this kind of prior knowledge are few and far between.

**Have a go at Roman to Integer first**

The problem of converting a [Roman Numeral to an Integer](https://leetcode.com/problems/roman-to-integer/) is simpler. Therefore, we suggest that you have a go at it first if you're finding this question difficult. This will allow you to become more familiar with the concept of Roman Numerals without the "ambiguity" issue that comes up in converting an integer to a Roman Numeral. When converting a Roman Numeral to an integer, there's only one sensible conversion.

**Roman Numeral Symbols**

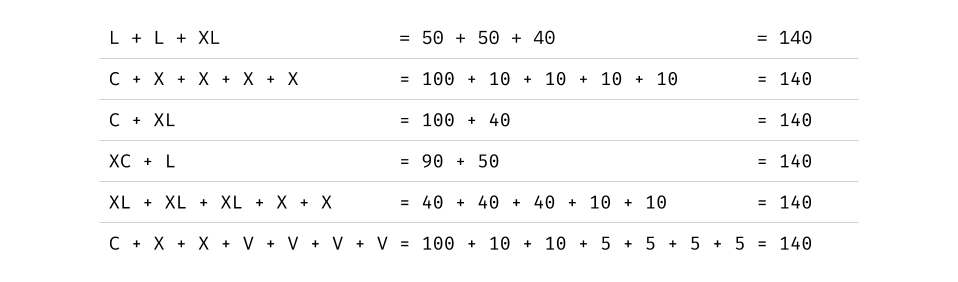
Roman Numerals are made with 7 single-letter symbols, each with its own value. Additionally, the subtractive rules (as explained in the problem description) give an additional 6 symbols. This gives us a total of 13 unique symbols (each symbol is made of either 1 letter or 2).



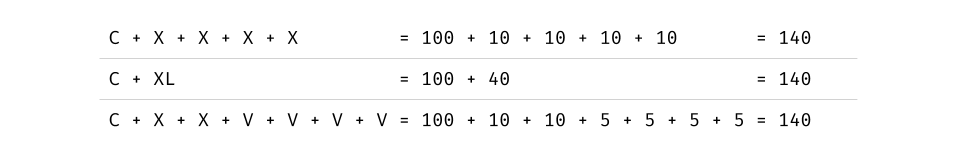
An integer is represented as a Roman Numeral by finding symbols that add to its value.

**Handling Ambiguity**

One thing that can be a bit confusing if you're not familiar with Roman Numerals is knowing which representation is the "correct" one for a particular integer. For example, consider these possible ways of representing 140. Which of these is correct?



**The system we use to decide** is to select the representation with the largest possible symbols, working from left to right. For example, the representations above with the largest symbol at the start are the ones starting with C.



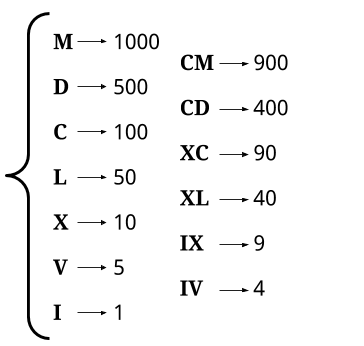
To decide which of these to go with, we look at the next symbol. Two of them have an X, which is worth 10, and one of them has an XL, which is worth 40. Because the XL is worth more, we go with that representation. Therefore, the representation for 140 is CXL.

This definition of Roman Numerals is, these days, the "most accepted". Interestingly, it still isn't an absolute standard, and throughout history, there have been many variants. If you're interested in math and history, we recommend checking out the [Wikipedia article](https://en.wikipedia.org/wiki/Roman_numerals) for your own interest.

#### **Approach 1: Greedy**

**Intuition**

Representing a given integer as a Roman Numeral requires finding a sequence of the above 13 symbols, where their corresponding values add up to the integer. This sequence must be in order from largest to smallest, based on symbol value. To remind you, these are the symbol values.



As explained in the overview, the representation should use the largest possible symbols, working from the left. Therefore, it makes sense to use a **Greedy** algorithm. A Greedy algorithm is an algorithm that makes the best possible decision at the current time; in this case taking out the largest possible symbol it can.

So to represent a given integer, we look for the largest symbol that fits into it. We subtract that, and then look for the largest symbol that fits into the remainder, and so on until the remainder is 0. Each of the symbols we take out are appended onto the output Roman Numeral string.

For example, suppose we need to make the number 671.

The largest symbol that fits into 671 is D (which is worth 500). The next symbol up, CM, is worth 900 and so is too big to fit. Therefore, we now have the following.

Roman Numeral so far: D

Integer remainder: 671 - 500 = 171

We now repeat the process with 171. The largest symbol that fits into it is C (worth 100).

Roman Numeral so far: DC

Integer remainder: 171 - 100 = 71

Repeating this with 71, we find the largest symbol that fits in is L (worth 50).

Roman Numeral so far: DCL

Integer remainder: 71 - 50 = 21

For 21, the largest symbol that fits in is X (worth 10).

Roman Numeral so far: DCLX

Integer remainder: 21 - 10 = 11

For 11, the largest symbol that fits in is again X.

Roman Numeral so far: DCLXX

Integer remainder: 11 - 10 = 1

Finally, the 1 is represented with a I and we're done.

Roman Numeral so far: DCLXXI

Integer remainder: 1 - 1 = 0

In pseudocode, this algorithm is as follows.

define function to\_roman(integer):

roman\_numeral = ""

while integer is non-zero:

symbol = biggest valued symbol that fits into integer

roman\_numeral = concat roman\_numeral and symbol

integer = integer - value of symbol

return roman\_numeral

The cleanest way to implement this in code is to loop over each symbol, from largest to smallest, checking how many copies of the current symbol fit into the remaining integer.

define function to\_roman(integer):

roman\_numeral = ""

for each symbol from largest to smallest:

if value of symbol is greater than integer:

continue

symbol\_count = number of times symbol value fits into integer

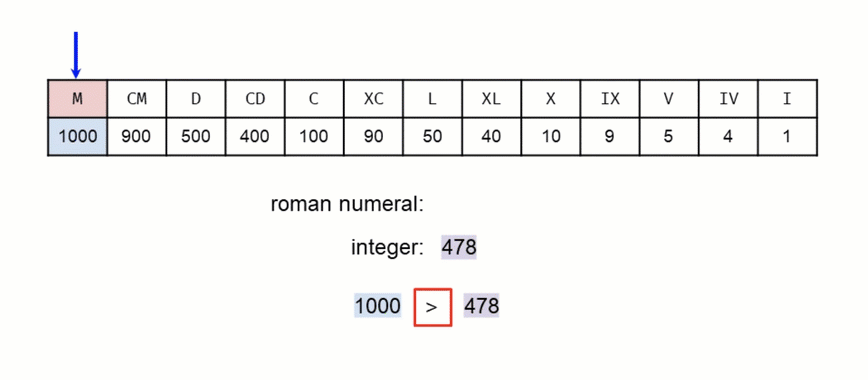
repeat symbol\_count times:

roman\_numeral = concat roman\_numeral and symbol

integer = integer - (value of symbol \* symbol\_count)

return roman\_numeral

Here's an animation showing this algorithm run on the number 478.



|  |
| --- |
| int[] values = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};  String[] symbols = {"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};  public String intToRoman(int num) {  StringBuilder sb = new StringBuilder();  // Loop through each symbol, stopping if num becomes 0.  for (int i = 0; i < values.length && num >= 0; i++) {  // Repeat while the current symbol still fits into num.  while (values[i] <= num) {  num -= values[i];  sb.append(symbols[i]);  }  }  return sb.toString();  } |

**Complexity Analysis**

* Time complexity : O(1)*O*(1).

As there is a finite set of roman numerals, there is a hard upper limit on how many times the loop can iterate. This upper limit is 15 times, and it occurs for the number 3888, which has a representation of MMMDCCCLXXXVIII. Therefore, we say the time complexity is constant, i.e. O(1)*O*(1).

* Space complexity : O(1)*O*(1).

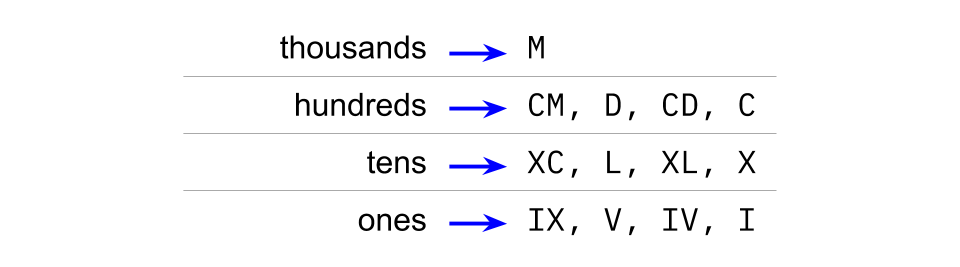
The amount of memory used does not change with the size of the input integer, and is therefore constant.

#### **Approach 2: Hardcode Digits**

**Intuition**

Please don't panic and assume you need to memorize the values in this approach. The first approach should be fine, and in-fact has the added bonus of being more flexible if we were to extend the Roman Numeral symbol set to have symbols over 1000. This second approach is only included for completeness. Do try to understand how we derived this approach, though.

An interesting observation that can be made is that each of the digits in the integer's decimal representation can be treated independently when converting the integer into a Roman Numeral. Notice that all of the symbols can be split into groups based on their highest factor out of 1000, 100, 10, and 1.

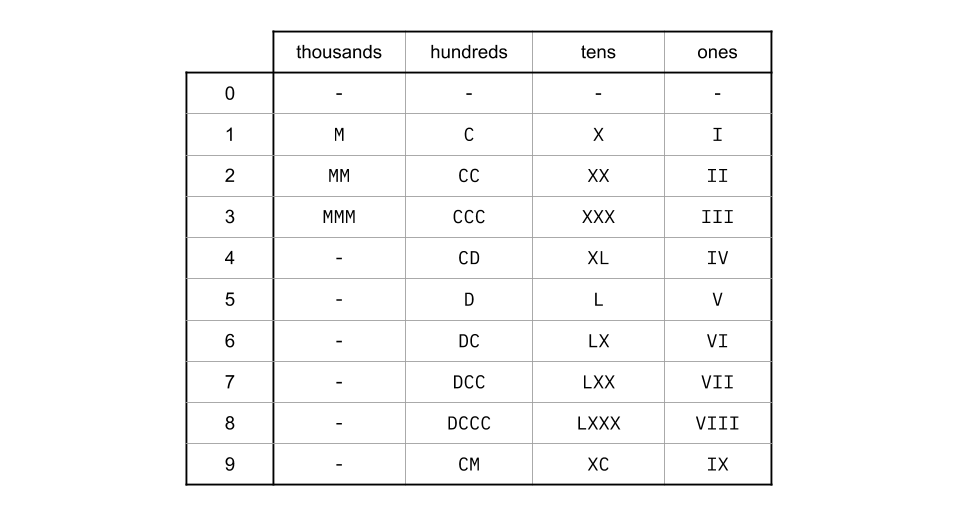


While the number is at least 1000, an M (1000) will be appended to the output and 1000 will be subtracted from the integer. The other symbols won't even be considered until the number is below 1000. Additionally, the M (1000)s cannot represent any lower part of the number. Therefore, we can represent the thousands digit of the integer entirely with M (1000)s.

Now, assume we have a remainder of between 100 and 999. The next symbols considered are those in the hundreds row. The highest symbol that could fit in right now is CM (900), and the lowest is C (100). None of the symbols in this range can possibly modify the tens or ones. As long as the remainder is still above 100, we can still take at least C (100) out of it. This means that we'll only be subtracting symbols from the hundreds row for as long as the number is at least 100.

The same argument applies for the tens, and then the ones.

We can, therefore, work out what the representation for each digit, in each place, is. There are only 34 of them; 0, 1, 2, 3 and 4 for the thousands column, and 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 for each of the hundreds, tens, and ones. So with a pencil, paper, and some patience, you can hopefully work out the representation for each of these possibilities and hardcode them. Then, converting an integer to a Roman Numeral will require breaking the integer into digits and appending the relevant representation for each digit.



Getting each digit of the number can be done using the modulus and division operators. The division operator removes the digits below the place we want, and the modulus operator removes the digits from above. This simply leaves the digit we want.

thousands\_digit = integer / 1000

hundreds\_digit = (integer % 1000) / 100

tens\_digit = (integer % 100) / 10

ones\_digit = integer % 10

Then, we can simply look these up in the hardcoded table, and append the results together!

**Algorithm**

The cleanest way to go about it in code is to have 4 separate arrays; one for each place value. Then, extract the digits, look up their symbols in the relevant array, and append them all together.

|  |
| --- |
| public String intToRoman(int num) {    String[] thousands = {"", "M", "MM", "MMM"};  String[] hundreds = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};  String[] tens = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};  String[] ones = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};    return thousands[num / 1000] + hundreds[num % 1000 / 100] + tens[num % 100 / 10] + ones[num % 10];  } |

**Complexity Analysis**

* Time complexity : O(1)*O*(1).

The same number of operations is done, regardless of the size of the input. Therefore, the time complexity is constant.

* Space complexity : O(1)*O*(1).

While we have Arrays, they are the same size, *regardless of the size of the input*. Therefore, they are constant for the purpose of space-complexity analysis.

The downside of this approach is that it is inflexible if Roman Numerals were to be extended (which is an interesting follow-up question). For example, what if we said the symbol H now represents 5000, and P now represents 10000, allowing us to represent numbers up to 39999? Approach 1 will be a lot quicker to modify, as you simply need to add these 2 values to the code without doing any calculations. But for Approach 2, you'll need to calculate and hardcode ten new representations. What if we then added symbols to be able to go up to 399,999,999? Approach 2 becomes more and more difficult to manage, the more symbols we add.

**3Sum Closest**

Given an array nums of *n* integers and an integer target, find three integers in nums such that the sum is closest to target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

**Example 1:**

**Input:** nums = [-1,2,1,-4], target = 1

**Output:** 2

**Explanation:** The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

**Constraints:**

* 3 <= nums.length <= 10^3
* -10^3 <= nums[i] <= 10^3
* -10^4 <= target <= 10^4

## Solution

This problem is a variation of [3Sum](https://leetcode.com/articles/3sum/). The main difference is that the sum of a triplet is not necessarily equal to the target. Instead, the sum is in some relation with the target, which is closest to the target for this problem. In that sense, this problem shares similarities with [3Sum Smaller](https://leetcode.com/articles/3sum-smaller/).

Before jumping in, let's check solutions for the similar problems:

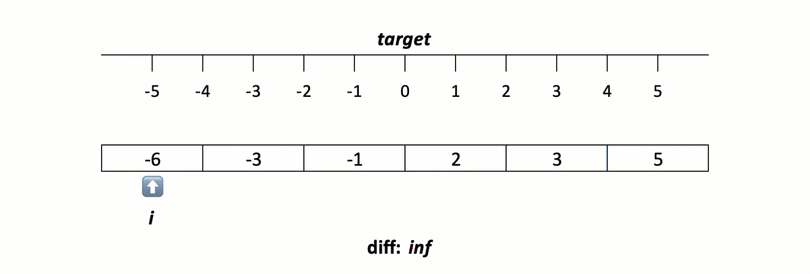
1. [3Sum](https://leetcode.com/articles/3sum/) fixes one number and uses either the two pointers pattern or a hash set to find complementary pairs. Thus, the time complexity is \mathcal{O}(n^2)O(*n*2).
2. [3Sum Smaller](https://leetcode.com/articles/3sum-smaller/), similarly to 3Sum, uses the two pointers pattern to enumerate smaller pairs. Note that we cannot use a hash set here because we do not have a specific value to look up.

For the same reason as for 3Sum Smaller, we cannot use a hash set approach here. So, we will focus on the two pointers pattern and shoot for \mathcal{O}(n^2)O(*n*2) time complexity as the best conceivable runtime (BCR).

#### **Approach 1: Two Pointers**

The two pointers pattern requires the array to be sorted, so we do that first. As our BCR is \mathcal{O}(n^2)O(*n*2), the sort operation would not change the overall time complexity.

In the sorted array, we process each value from left to right. For value v, we need to find a pair which sum, ideally, is equal to target - v. We will follow the same two pointers approach as for 3Sum, however, since this 'ideal' pair may not exist, we will track the smallest absolute difference between the sum and the target. The two pointers approach naturally enumerates pairs so that the sum moves toward the target.



**Algorithm**

1. Initialize the minimum difference diff with a large value.
2. Sort the input array nums.
3. Iterate through the array:
   * For the current position i, set lo to i + 1, and hi to the last index.
   * While the lo pointer is smaller than hi:
     + Set sum to nums[i] + nums[lo] + nums[hi].
     + If the absolute difference between sum and target is smaller than the absolute value of diff:
       - Set diff to target - sum.
     + If sum is less than target, increment lo.
     + Else, decrement hi.
   * If diff is zero, break from the loop.
4. Return the value of the closest triplet, which is target - diff.

|  |
| --- |
| public int threeSumClosest(int[] nums, int target) {  int diff = Integer.MAX\_VALUE, sz = nums.length;  Arrays.sort(nums);  for (int i = 0; i < sz && diff != 0; ++i) {  int lo = i + 1, hi = sz - 1;  while (lo < hi) {  int sum = nums[i] + nums[lo] + nums[hi];  if (Math.abs(target - sum) < Math.abs(diff))  diff = target - sum;  if (sum < target)  ++lo;  else  --hi;  }  }  return target - diff;  } |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(n^2)O(*n*2). We have outer and inner loops, each going through n*n* elements.

Sorting the array takes \mathcal{O}(n\log{n})O(*n*log*n*), so overall complexity is \mathcal{O}(n\log{n} + n^2)O(*n*log*n*+*n*2). This is asymptotically equivalent to \mathcal{O}(n^2)O(*n*2).

* Space Complexity: from \mathcal{O}(\log{n})O(log*n*) to \mathcal{O}(n)O(*n*), depending on the implementation of the sorting algorithm.

#### **Approach 2: Binary Search**

We can adapt the [3Sum Smaller: Binary Search](https://leetcode.com/articles/3sum-smaller/#approach-2-binary-search-accepted) approach to this problem.

In the two pointers approach, we fix one number and use two pointers to enumerate pairs. Here, we fix two numbers, and use a binary search to find the third complement number. This is less efficient than the two pointers approach, however, it could be more intuitive to come up with.

Note that we may not find the exact complement number, so we check the difference between the complement and two numbers: the next higher and the previous lower. For example, if the complement is 42, and our array is [-10, -4, 15, 30, 60], the next higher is 60 (so the difference is -18), and the previous lower is 30 (and the difference is 12).

**Algorithm**

1. Initialize the minimum difference diff with a large value.
2. Sort the input array nums.
3. Iterate through the array (outer loop):
   * For the current position i, iterate through the array starting from j = i + 1 (inner loop):
     + Binary-search for complement (target - nums[i] - nums[j]) in the rest of the array.
     + For the next higher value, check its absolute difference with complement against diff.
     + For the previous lower value, check its absolute difference with complement against diff.
     + Update diff based on the smallest absolute difference.
   * If diff is zero, break from the loop.
4. Return the value of the closest triplet, which is target - diff.

|  |
| --- |
| public int threeSumClosest(int[] nums, int target) {  int diff = Integer.MAX\_VALUE, sz = nums.length;  Arrays.sort(nums);  for (int i = 0; i < sz && diff != 0; ++i) {  for (int j = i + 1; j < sz - 1; ++j) {  int complement = target - nums[i] - nums[j];  var idx = Arrays.binarySearch(nums, j + 1, sz - 1, complement);  int hi = idx >= 0 ? idx : ~idx, lo = hi - 1;  if (hi < sz && Math.abs(complement - nums[hi]) < Math.abs(diff))  diff = complement - nums[hi];  if (lo > j && Math.abs(complement - nums[lo]) < Math.abs(diff))  diff = complement - nums[lo];  }  }  return target - diff;  } |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(n^2\log{n})O(*n*2log*n*). Binary search takes \mathcal{O}(\log{n})O(log*n*), and we do it n*n* times in the inner loop. Since we are going through n*n* elements in the outer loop, the overall complexity is \mathcal{O}(n^2\log{n})O(*n*2log*n*).
* Space Complexity: from \mathcal{O}(\log{n})O(log*n*) to \mathcal{O}(n)O(*n*), depending on the implementation of the sorting algorithm.

#### **Further Thoughts**

3Sum is a well-known problem with many variations and its own [Wikipedia page](https://en.wikipedia.org/wiki/3SUM).

For an interview, we recommend focusing on the Two Pointers approach above. It's easier to get it right and adapt for other variations of 3Sum. Interviewers love asking follow-up problems like [3Sum Smaller](https://leetcode.com/articles/3sum-smaller/)!

If an interviewer asks you whether you can achieve \mathcal{O}(1)O(1) memory complexity, you can use the selection sort instead of a built-in sort in the Two Pointers approach. It will make it a bit slower, though the overall time complexity will be still \mathcal{O}(n^2)O(*n*2).

**Compare Version Numbers**

Given two version numbers, version1 and version2, compare them.

Version numbers consist of **one or more revisions** joined by a dot '.'. Each revision consists of **digits** and may contain leading **zeros**. Every revision contains **at least one character**. Revisions are **0-indexed from left to right**, with the leftmost revision being revision 0, the next revision being revision 1, and so on. For example 2.5.33 and 0.1 are valid version numbers.

To compare version numbers, compare their revisions in **left-to-right order**. Revisions are compared using their **integer value ignoring any leading zeros**. This means that revisions 1 and 001 are considered **equal**. If a version number does not specify a revision at an index, then **treat the revision as 0**. For example, version 1.0 is less than version 1.1 because their revision 0s are the same, but their revision 1s are 0 and 1 respectively, and 0 < 1.

*Return the following:*

* If version1 < version2, return -1.
* If version1 > version2, return 1.
* Otherwise, return 0.

**Example 1:**

**Input:** version1 = "1.01", version2 = "1.001"

**Output:** 0

**Explanation:** Ignoring leading zeroes, both "01" and "001" represent the same integer "1".

**Example 2:**

**Input:** version1 = "1.0", version2 = "1.0.0"

**Output:** 0

**Explanation:** version1 does not specify revision 2, which means it is treated as "0".

**Example 3:**

**Input:** version1 = "0.1", version2 = "1.1"

**Output:** -1

**Explanation:** version1's revision 0 is "0", while version2's revision 0 is "1". 0 < 1, so version1 < version2.

**Example 4:**

**Input:** version1 = "1.0.1", version2 = "1"

**Output:** 1

**Example 5:**

**Input:** version1 = "7.5.2.4", version2 = "7.5.3"

**Output:** -1

**Constraints:**

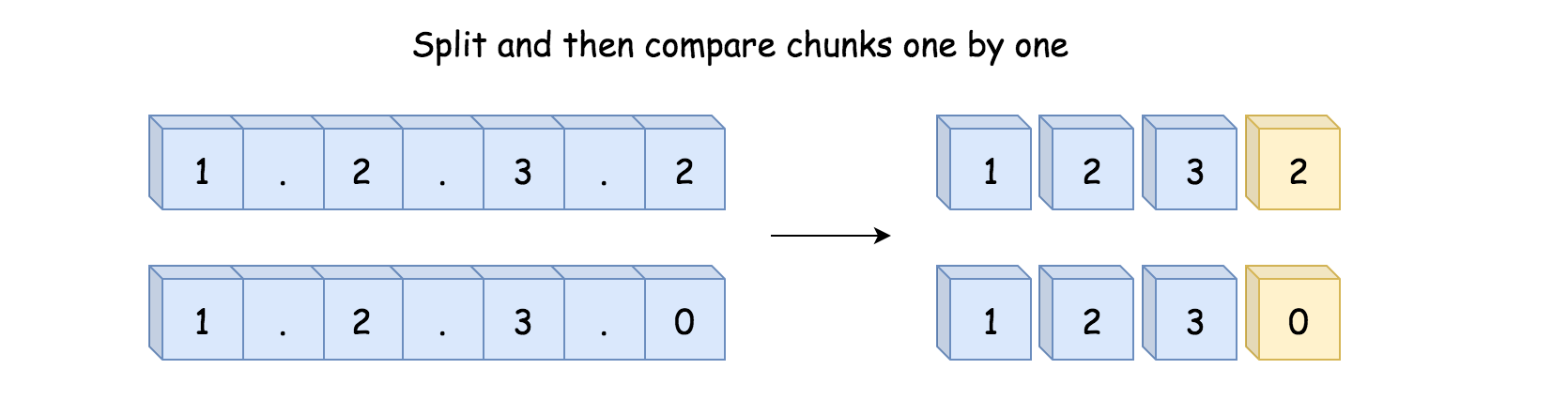
* 1 <= version1.length, version2.length <= 500
* version1 and version2 only contain digits and '.'.
* version1 and version2 **are valid version numbers**.
* All the given revisions in version1 and version2 can be stored in a **32-bit integer**.

Solution

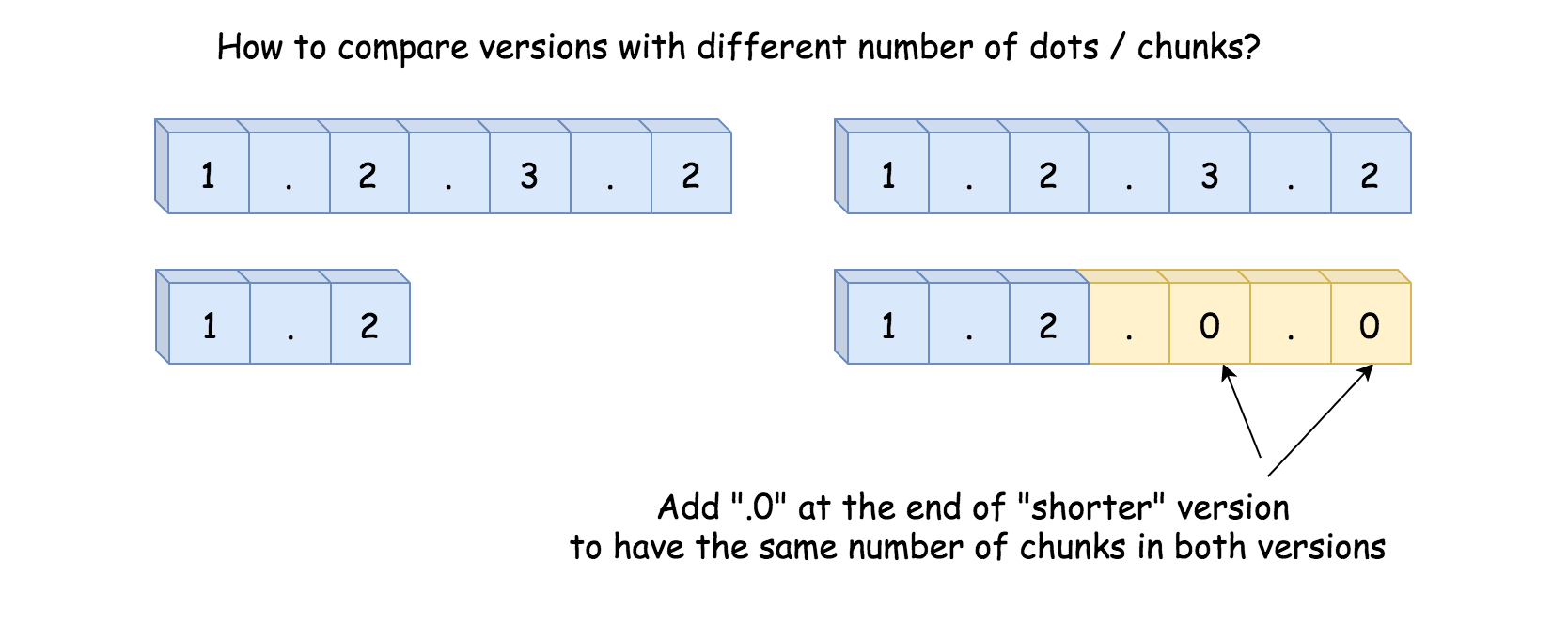
Approach 1: Split + Parse, Two Pass

**Intuition**

The first idea is to split both strings by dot character into chunks and then compare the chunks one by one.



That works fine if the number of chunks is the same for both versions. If not, we need to pad the shorter string by adding .0 at the end of the string with less chunks one or several times, so that the number of chunks will be the same.



**Algorithm**

* Split both strings by dot character into two arrays.
* Iterate over the longest array and compare chunks one by one. If one of the arrays is over, virtually add as many zeros as needed to continue the comparison with the longer array.
  + If two chunks are not equal, return 1 or -1.
* If we're here, the versions are equal. Return 0.

**Implementation**

|  |
| --- |
| class Solution {  public int compareVersion(String version1, String version2) {  String[] nums1 = version1.split("\\.");  String[] nums2 = version2.split("\\.");  int n1 = nums1.length, n2 = nums2.length;  // compare versions  int i1, i2;  for (int i = 0; i < Math.max(n1, n2); ++i) {  i1 = i < n1 ? Integer.parseInt(nums1[i]) : 0;  i2 = i < n2 ? Integer.parseInt(nums2[i]) : 0;  if (i1 != i2) {  return i1 > i2 ? 1 : -1;  }  }  // the versions are equal  return 0;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N + M + \max(N, M))O(*N*+*M*+max(*N*,*M*)), where N*N* and M*M* are lengths of input strings.
* Space complexity : \mathcal{O}(N + M)O(*N*+*M*) to store arrays nums1 and nums2.

#### **Approach 2: Two Pointers, One Pass**

**Intuition**

Rather than splitting the string all at once with the split() function, we could also split the string **on the fly**, through which we only need to iterate through the revisions once.

The idea is that we split the string chunk by chunk, i.e. each trunk represents a revision in the version number. The moment we retrieve a trunk from each string, we then compare them.

In this way, one could move along both strings in parallel, retrieve and compare corresponding chunks. Once both strings are parsed, the comparison is done as well.

As a result, the process can be done in a **single** pass.

**Algorithm**

First, we define a function named get\_next\_chunk(version, n, p), which is to retrieve the next chunk in the string.

This function takes three arguments: the input string version, its length n, and a pointer p set to the first character of chunk to retrieve. It returns an integer chunk in-between the pointer p and the next dot. To help with the iteration, it returns as well a pointer set to the first character of the next chunk.

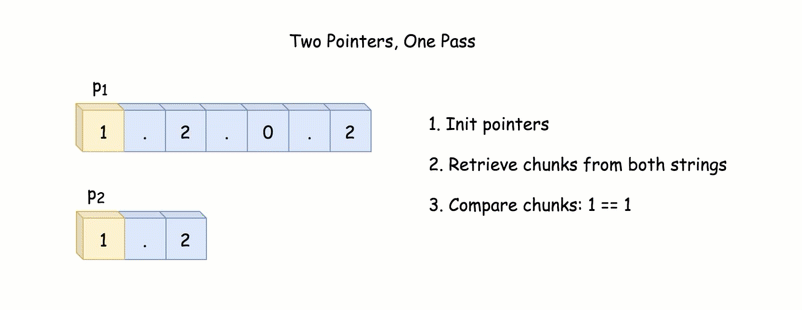
Here is how one could solve the problem using this function:

* Set a pointer p1 pointed to the beginning of string version1 and a pointer p2 to the beginning of string version2: p1 = p2 = 0.
* Iterate over both strings in parallel. While p1 < n1 or p2 < n2:
  + Retrieve the next chunk i1 from string version1 and next chunk i2 from string version2 using the above-defined get\_next\_chunk function.
  + Compare i1 and i2. If they are not equal, return 1 or -1.
* If we're here, the versions are equal. Return 0.

Now let's implement our get\_next\_chunk(version, n, p) function:

* The beginning of chunk is marked by the pointer p. If p is set to the end of string, the string is already parsed. To continue the comparison, let's add a virtual .0 at the end of this string by returning 0.
* If p is not at the end of string, move the pointer p\_end along the string to find the end of chunk.
* Return the chunk version.substring(p, p\_end).

**Implementation**



|  |
| --- |
| class Solution {  public Pair<Integer, Integer> getNextChunk(String version, int n, int p) {  // if pointer is set to the end of string  // return 0  if (p > n - 1) {  return new Pair(0, p);  }  // find the end of chunk  int i, pEnd = p;  while (pEnd < n && version.charAt(pEnd) != '.') {  ++pEnd;  }  // retrieve the chunk  if (pEnd != n - 1) {  i = Integer.parseInt(version.substring(p, pEnd));  } else {  i = Integer.parseInt(version.substring(p, n));  }  // find the beginning of next chunk  p = pEnd + 1;  return new Pair(i, p);  }  public int compareVersion(String version1, String version2) {  int p1 = 0, p2 = 0;  int n1 = version1.length(), n2 = version2.length();  // compare versions  int i1, i2;  Pair<Integer, Integer> pair;  while (p1 < n1 || p2 < n2) {  pair = getNextChunk(version1, n1, p1);  i1 = pair.getKey();  p1 = pair.getValue();  pair = getNextChunk(version2, n2, p2);  i2 = pair.getKey();  p2 = pair.getValue();  if (i1 != i2) {  return i1 > i2 ? 1 : -1;  }  }  // the versions are equal  return 0;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(\max(N, M))O(max(*N*,*M*)), where N*N* and M*M* are lengths of the input strings respectively. It's a one-pass solution.
* Space complexity : \mathcal{O}(\max(N, M))O(max(*N*,*M*)).
  + Despite the fact that we did not keep arrays of revision numbers, we still need some additional space to store a substring of the input string for integer conversion. In the worst case, the substring could be of the original string as well.

**Most Common Word**

Given a paragraph and a list of banned words, return the most frequent word that is not in the list of banned words.  It is guaranteed there is at least one word that isn't banned, and that the answer is unique.

Words in the list of banned words are given in lowercase, and free of punctuation.  Words in the paragraph are not case sensitive.  The answer is in lowercase.

**Example:**

**Input:**

paragraph = "Bob hit a ball, the hit BALL flew far after it was hit."

banned = ["hit"]

**Output:** "ball"

**Explanation:**

"hit" occurs 3 times, but it is a banned word.

"ball" occurs twice (and no other word does), so it is the most frequent non-banned word in the paragraph.

Note that words in the paragraph are not case sensitive,

that punctuation is ignored (even if adjacent to words, such as "ball,"),

and that "hit" isn't the answer even though it occurs more because it is banned.

**Note:**

* 1 <= paragraph.length <= 1000.
* 0 <= banned.length <= 100.
* 1 <= banned[i].length <= 10.
* The answer is unique, and written in lowercase (even if its occurrences in paragraph may have uppercase symbols, and even if it is a proper noun.)
* paragraph only consists of letters, spaces, or the punctuation symbols !?',;.
* There are no hyphens or hyphenated words.
* Words only consist of letters, never apostrophes or other punctuation symbols.

## Solution

#### **Overview**

This problem is a good exercise to brush up one's skills on string manipulation.

The String data type is almost omnipresent in all programming languages. However, each language has its own implementation of String type, as well as various APIs for string manipulation. For instance, String is mutable in C++, while immutable in Python and Java.

The problem is not difficult. But due to the diversity of String type and string manipulation APIs, one could come up many different solutions.

Here we give two general approaches in the following sections.

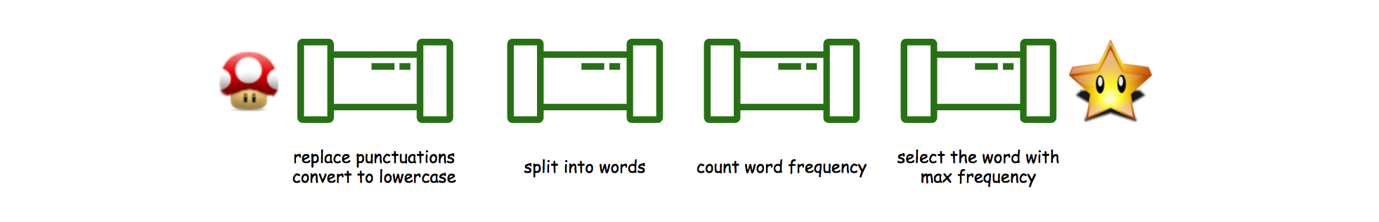
* In one approach, we will construct a pipeline to process strings in several stages, where naturally each string would be traversed for several times.
* In another approach, we will traverse the input string once and only once, on the character base and do the processing ***on-the-fly***.

#### **Approach 1: String Processing in Pipeline**

**Intuition**

We can solve the problem by breaking it into a series of sequential tasks. Each task functions like a stage in a pipeline, which takes the input from the previous stage and then channels its output to the next stage.

More specifically, for this problem, we could break it down into the following stages:



1. We replace all the punctuations with spaces and at the same time convert each letter to its lowercase. One could also accomplish this in two stages. Here we merge them together in one stage.
2. We split the output in the above step into words, with the separator of spaces.
3. We then iterate through the words to count the appearance of each unique word, excluding the words from the banned list.
4. With the hashmap of {word->count}, we then walk through all the items to find the word with the highest frequency.

**Algorithm**

Following the stages we explained before, here are some sample implementations.

|  |
| --- |
| class Solution {  public String mostCommonWord(String paragraph, String[] banned) {  // 1). replace the punctuations with spaces,  // and put all letters in lower case  String normalizedStr = paragraph.replaceAll("[^a-zA-Z0-9 ]", " ").toLowerCase();  // 2). split the string into words  String[] words = normalizedStr.split("\\s+");  Set<String> bannedWords = new HashSet();  for (String word : banned)  bannedWords.add(word);  Map<String, Integer> wordCount = new HashMap();  // 3). count the appearance of each word, excluding the banned words  for (String word : words) {  if (!bannedWords.contains(word))  wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);  }  // 4). return the word with the highest frequency  return Collections.max(wordCount.entrySet(), Map.Entry.comparingByValue()).getKey();  }  } |

**Complexity Analysis**

Let N*N* be the number of characters in the input string and M*M* be the number of characters in the banned list.

* Time Complexity: \mathcal{O}(N + M)O(*N*+*M*).
  + It would take \mathcal{O}(N)O(*N*) time to process each stage of the pipeline as we built.
  + In addition, we built a set out of the list of banned words, which would take \mathcal{O}(M)O(*M*) time.
  + Hence, the overall time complexity of the algorithm is \mathcal{O}(N + M)O(*N*+*M*).
* Space Complexity: \mathcal{O}(N + M)O(*N*+*M*).
  + We built a hashmap to count the frequency of each unique word, whose space would be of \mathcal{O}(N)O(*N*).
  + Similarly, we built a set out of the banned word list, which would consume additional \mathcal{O}(M)O(*M*) space.
  + Therefore, the overall space complexity of the algorithm is \mathcal{O}(N + M)O(*N*+*M*).

#### **Approach 2: Character Processing in One-Pass**

**Intuition**

With the approach of String manipulation pipeline, it is clear and easy to debug, since we could locate and inspect each stage if anything goes wrong.

However, one might argue that it is probably not the most efficient way to solve the problem, since we scan the input string multiple times.

Indeed, it is possible to process the input string once and only once to accomplish the tasks.

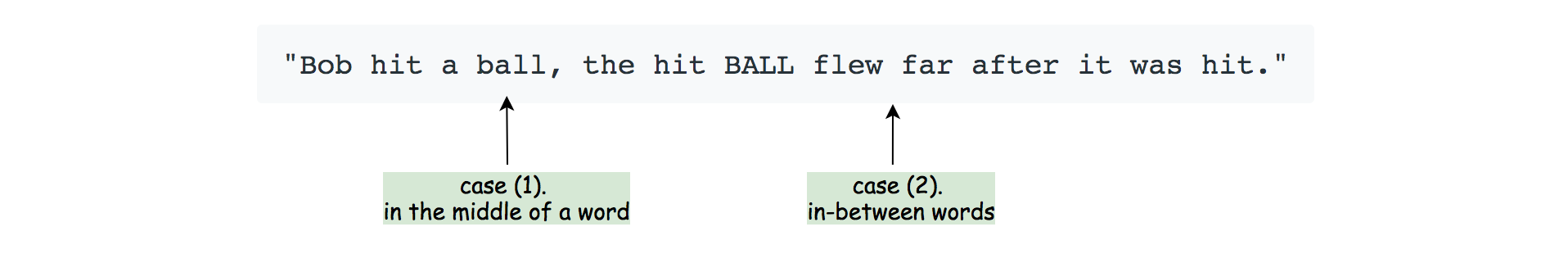
We could iterate through the string character by character, and do the processing ***on-the-fly***, rather than delaying the processing to the latter stages of the pipeline.

The idea is that we consume the input string on the character base. At the moment we reach the end of one word, we can then start to perform the word-based logics such as checking if the word is in the banned list, updating the frequency of the word and also updating the most frequent word we've seen so far etc.

**Algorithm**

We could implement the algorithm in one single loop, over the characters of the input string.

* At each iteration, the character is either of letter (maybe digit), or punctuation or space in other cases.



* Further more, we could divide it into the following two cases:
  + **Case (1):** we are in the middle of a word.
  + **Case (2):** we in in-between the words, e.g. punctuations between the words or at the end of the paragraph.
* We then can organize the logics into the above two cases.
  + In case (1), we simply append the character into the word buffer.
  + In case (2), we do the rest of the logics, as follows:
    - check if the word is enlisted in the banned list.
    - if not, update the frequency of the word.
    - update the most common word that we've seen so far.

|  |
| --- |
| class Solution {  public String mostCommonWord(String paragraph, String[] banned) {  Set<String> bannedWords = new HashSet();  for (String word : banned)  bannedWords.add(word);  String ans = "";  int maxCount = 0;  Map<String, Integer> wordCount = new HashMap();  StringBuilder wordBuffer = new StringBuilder();  char[] chars = paragraph.toCharArray();  for (int p = 0; p < chars.length; ++p) {  char currChar = chars[p];  // 1). consume the characters in a word  if (Character.isLetter(currChar)) {  wordBuffer.append(Character.toLowerCase(currChar));  if (p != chars.length - 1)  // skip the rest of the processing  continue;  }  // 2). at the end of one word or at the end of paragraph  if (wordBuffer.length() > 0) {  String word = wordBuffer.toString();  // identify the maximum count while updating the wordCount table.  if (!bannedWords.contains(word)) {  int newCount = wordCount.getOrDefault(word, 0) + 1;  wordCount.put(word, newCount);  if (newCount > maxCount) {  ans = word;  maxCount = newCount;  }  }  // reset the buffer for the next word  wordBuffer = new StringBuilder();  }  }  return ans;  }  } |

**Complexity Analysis**

Let N*N* be the number of characters in the input string and M*M* be the number of characters in the banned list.

* Time Complexity: \mathcal{O}(N + M)O(*N*+*M*).
  + We traverse each character in the input string once and only once. At each iteration, it takes constant time to perform the operations, except the operation that we build a new string out of the buffer. Excluding the cost of string-building out of the iteration, we can consider the cost of iterations as \mathcal{O}(N)O(*N*).
  + If we combine all the string-building operations all together, in total it would take another \mathcal{O}(N)O(*N*) time.
  + In addition, we built a set out of the list of banned words, which would take \mathcal{O}(M)O(*M*) time.
  + Hence, the overall time complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(N) + \mathcal{O}(M) = \mathcal{O}(N + M)O(*N*)+O(*N*)+O(*M*)=O(*N*+*M*).
* Space Complexity: \mathcal{O}(N + M)O(*N*+*M*).
  + We built a hashmap to count the frequency of each unique word, whose space would be of \mathcal{O}(N)O(*N*).
  + Similarly, we built a set out of the banned word list, which would consume additional \mathcal{O}(M)O(*M*) space.
  + Therefore, the overall space complexity of the algorithm is \mathcal{O}(N + M)O(*N*+*M*).

**Reorder Log Files**

You are given an array of logs. Each log is a space-delimited string of words, where the first word is the **identifier**.

There are two types of logs:

* **Letter-logs**: All words (except the identifier) consist of lowercase English letters.
* **Digit-logs**: All words (except the identifier) consist of digits.

Reorder these logs so that:

1. The **letter-logs** come before all **digit-logs**.
2. The **letter-logs** are sorted lexicographically by their contents. If their contents are the same, then sort them lexicographically by their identifiers.
3. The **digit-logs** maintain their relative ordering.

Return *the final order of the logs*.

**Example 1:**

**Input:** logs = ["dig1 8 1 5 1","let1 art can","dig2 3 6","let2 own kit dig","let3 art zero"]

**Output:** ["let1 art can","let3 art zero","let2 own kit dig","dig1 8 1 5 1","dig2 3 6"]

**Explanation:**

The letter-log contents are all different, so their ordering is "art can", "art zero", "own kit dig".

The digit-logs have a relative order of "dig1 8 1 5 1", "dig2 3 6".

**Example 2:**

**Input:** logs = ["a1 9 2 3 1","g1 act car","zo4 4 7","ab1 off key dog","a8 act zoo"]

**Output:** ["g1 act car","a8 act zoo","ab1 off key dog","a1 9 2 3 1","zo4 4 7"]

**Constraints:**

* 1 <= logs.length <= 100
* 3 <= logs[i].length <= 100
* All the tokens of logs[i] are separated by a **single** space.
* logs[i] is guaranteed to have an identifier and at least one word after the identifier.

## Solution

#### **Overview**

First of all, let us put aside the debate whether this problem is an easy or medium one. The problem is a good exercise to practice the technique of **custom sort** in different languages.

The idea of custom sort is that we don't have to rewrite a sorting algorithm every time we have a different ***sorting criteria*** among the elements.

Each language provides certain interface that allows us to **customize** the sorting criteria of the sorting functions, so that we can reuse the implementation of sorting in different scenarios.

In this article, we will present two ways to specify the sorting order, namely by **comparator** and by **sorting key**.

#### **Approach 1: Comparator**

**Intuition**

Given a list of elements [e\_1, e\_2, e\_3][*e*1​,*e*2​,*e*3​], regardless of the content of the elements, the first way to specify the order among the elements is to define the **pairwise** << ("less than") **relationship** globally.

For instance, for the above example, we could define the **relationships** as e\_3 < e\_2, \space e\_2 < e\_1*e*3​<*e*2​, *e*2​<*e*1​. Then if we are asked to sort the list in the ascending order, the result would be [e\_3, e\_2, e\_1][*e*3​,*e*2​,*e*1​].

**Note:** normally we should define all pairwise relationships among all elements, but due to the transitive property, we omit certain relationships that can be deduced from others, e.g. (e\_3 < e\_2, e\_2 < e\_1) \to (e\_1 < e\_3)(*e*3​<*e*2​,*e*2​<*e*1​)→(*e*1​<*e*3​)

If we ever change the order, e.g. e\_1 < e\_3, \space e\_3 < e\_2*e*1​<*e*3​, *e*3​<*e*2​, the final sorted result would be changed accordingly, i.e. [e\_1, e\_3, e\_2][*e*1​,*e*3​,*e*2​].

**Algorithm**

The above pairwise "less than" relationship is also known as **comparator** in Java, which is a function object that helps the sorting functions to determine the orders among a collection of elements.

We show the [definition of the comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) interface as follows:

int compare(T o1, T o2) {

if (o1 < o2)

return -1;

else if (o1 == o2)

return 0;

else // o1 > o2

return 1;

}

As we discussed before, once we define the pairwise relationship among the elements in a collection, the **total order** of the collection is then fixed.

Now, what we need to do is to define our own proper **comparator** according to the description of the problem. We can translate the problem into the following rules:

* 1). The letter-logs should be prioritized above all digit-logs.
* 2). Among the letter-logs, we should further sort them firstly based on their **contents**, and then on their **identifiers** if the contents are identical.
* 3). Among the digit-logs, they should remain in the same order as they are in the collection.

One can then go ahead and implement the comparator based on the above rules. Here is an example.

|  |
| --- |
| class Solution {  public String[] reorderLogFiles(String[] logs) {  Comparator<String> myComp = new Comparator<String>() {  @Override  public int compare(String log1, String log2) {  // split each log into two parts: <identifier, content>  String[] split1 = log1.split(" ", 2);  String[] split2 = log2.split(" ", 2);  boolean isDigit1 = Character.isDigit(split1[1].charAt(0));  boolean isDigit2 = Character.isDigit(split2[1].charAt(0));  // case 1). both logs are letter-logs  if (!isDigit1 && !isDigit2) {  // first compare the content  int cmp = split1[1].compareTo(split2[1]);  if (cmp != 0)  return cmp;  // logs of same content, compare the identifiers  return split1[0].compareTo(split2[0]);  }  // case 2). one of logs is digit-log  if (!isDigit1 && isDigit2)  // the letter-log comes before digit-logs  return -1;  else if (isDigit1 && !isDigit2)  return 1;  else  // case 3). both logs are digit-log  return 0;  }  };  Arrays.sort(logs, myComp);  return logs;  }  } |

**Stable Sort**

One might notice that in the above implementation one can find the logic that corresponds each of the rules, except the **Rule (3)**.

Indeed, we did not do anything explicitly to ensure the order imposed by the Rule (3).

The short answer is that the Rule (3) is ensured implicitly by an important property of sorting algorithms, called [**stability**](https://en.wikipedia.org/wiki/Sorting_algorithm#Stability).

It is stated as "stable sorting algorithms sort equal elements in the same order that they appear in the input."

Not all sort algorithms are stable, e.g. **merge sort** is stable.

The Arrays.sort() interface that we used is stable, as one can find in the [specification](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html).

Therefore, the Rule (3) is implicitly respected thanks to the stability of the sorting algorithm that we used.

**Complexity Analysis**

Let N*N* be the number of logs in the list and M*M* be the maximum length of a single log.

* Time Complexity: \mathcal{O}(M \cdot N \cdot \log N)O(*M*⋅*N*⋅log*N*)
  + First of all, the time complexity of the Arrays.sort() is \mathcal{O}(N \cdot \log N)O(*N*⋅log*N*), as stated in the [API specification](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-byte:A-), which is to say that the compare() function would be invoked \mathcal{O}(N \cdot \log N)O(*N*⋅log*N*) times.
  + For each invocation of the compare() function, it could take up to \mathcal{O}(M)O(*M*) time, since we compare the contents of the logs.
  + Therefore, the overall time complexity of the algorithm is \mathcal{O}(M \cdot N \cdot \log N)O(*M*⋅*N*⋅log*N*).
* Space Complexity: \mathcal{O}(M \cdot \log N)O(*M*⋅log*N*)
  + For each invocation of the compare() function, we would need up to \mathcal{O}(M)O(*M*) space to hold the parsed logs.
  + In addition, since the implementation of Arrays.sort() is based on quicksort algorithm whose space complexity is \mathcal{O}(\log n)O(log*n*), assuming that the space for each element is \mathcal{O}(1)O(1)). Since each log could be of \mathcal{O}(M)O(*M*) space, we would need \mathcal{O}(M \cdot \log N)O(*M*⋅log*N*) space to hold the intermediate values for sorting.
  + In total, the overall space complexity of the algorithm is \mathcal{O}(M + M \cdot \log N) = \mathcal{O}(M \cdot \log N)O(*M*+*M*⋅log*N*)=O(*M*⋅log*N*).

#### **Approach 2: Sorting by Keys**

**Intuition**

Rather than defining pairwise relationships among all elements in a collection, the order of the elements can also be defined with **sorting keys**.

To illustrate the idea, let us first define a Student object as follows, which has three properties: name, grade, age.

class Student:

def \_\_init\_\_(self, name, grade, age):

self.name = name

self.grade = grade

self.age = age

student\_objects = [

Student('john', 'A', 15),

Student('jane', 'B', 12),

Student('dave', 'B', 10),

]

Now, if we are asked to sort the list of students by age in ascending order, we could simply use the age property of each student as the sorting key, as follows:

>>> sorted(student\_objects, key=lambda student: student.age)

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

Furthermore, the key could be a tuple of multiple keys, i.e. tuple(key\_1, key\_2, ... key\_n).

If two elements have the same value on key\_1, the comparison will carry on for the following keys, i.e. key\_2 ... key\_n.

As a result, if we are asked to sort the students first by the grade, then by the age, we can simply return the compound key (stduent.grade, student.age), as follows:

>>> sorted(student\_objects, key=lambda student: (student.grade, student.age))

[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

**Algorithm**

Given the above intuition, it should be clear that all we need is to translate the rules we defined before into a tuple of keys.

As a reminder, here are a list of the rules that we defined before, concerning the order of logs:

* 1). The letter-logs should be prioritized above all digit-logs.
* 2). Among the letter-logs, we should further sort them based on firstly on their **contents**, and then on their **identifiers** if the contents are identical.
* 3). Among the digit-logs, they should remain in the same order as they are in the collection.

To ensure the above order, we could define a tuple of 3 keys, (key\_1, key\_2, key\_3), as follows:

* key\_1: this key serves as a indicator for the type of logs. For the letter-logs, we could assign its key\_1 with 0, and for the digit-logs, we assign its key\_1 with 1. As we can see, thanks to the assigned values, the letter-logs would take the priority above the digit-logs.
* key\_2: for this key, we use the **content** of the letter-logs as its value, so that among the letter-logs, they would be further ordered based on their content, as required in the Rule (2).
* key\_3: similarly with the key\_2, this key serves to further order the letter-logs. We will use the **identifier** of the letter-logs as its value, so that for the letter-logs with the same content, we could further sort the logs based on its identifier, as required in the Rule (2).

**Note:** for the digit-logs, we don't need the key\_2 and key\_3. We can simply assign the None value to these two keys. As a result, the key value for all the digit-logs would be (1, None, None).

Finally, thanks to the **stability** of sorting algorithms, the elements with the same key value would remain the same order as in the original input. Therefore, the Rule (3) is ensured.

|  |
| --- |
| class Solution:  def reorderLogFiles(self, logs: List[str]) -> List[str]:  def get\_key(log):  \_id, rest = log.split(" ", maxsplit=1)  return (0, rest, \_id) if rest[0].isalpha() else (1, )  return sorted(logs, key=get\_key) |

**Complexity Analysis**

Let N*N* be the number of logs in the list and M*M* be the maximum length of a single log.

* Time Complexity: \mathcal{O}(M \cdot N \cdot \log N)O(*M*⋅*N*⋅log*N*)
  + The sorted() in Python is implemented with the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm whose time complexity is \mathcal{O}(N \cdot \log N)O(*N*⋅log*N*).
  + Since the keys of the elements are basically the logs itself, the comparison between two keys can take up to \mathcal{O}(M)O(*M*) time.
  + Therefore, the overall time complexity of the algorithm is \mathcal{O}(M \cdot N \cdot \log N)O(*M*⋅*N*⋅log*N*).
* Space Complexity: \mathcal{O}(M \cdot N)O(*M*⋅*N*)
  + First, we need \mathcal{O}(M \cdot N)O(*M*⋅*N*) space to keep the keys for the log.
  + In addition, the worst space complexity of the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm is \mathcal{O}(N)O(*N*), assuming that the space for each element is \mathcal{O}(1)O(1). Hence we would need \mathcal{O}(M \cdot N)O(*M*⋅*N*) space to hold the intermediate values for sorting.
  + In total, the overall space complexity of the algorithm is \mathcal{O}(M \cdot N + M \cdot N) = \mathcal{O}(M \cdot N)O(*M*⋅*N*+*M*⋅*N*)=O(*M*⋅*N*).

## Linked Lists

These are some of the must-practice linked list questions asked by Amazon. We recommend you practice all of these questions.

**Reverse Nodes in k-Group**

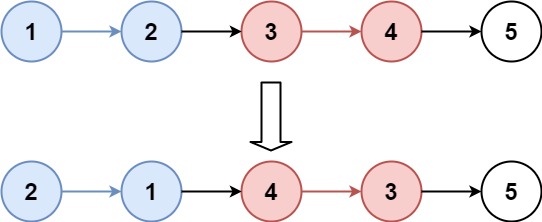
Given a linked list, reverse the nodes of a linked list *k* at a time and return its modified list.

*k* is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of *k* then left-out nodes, in the end, should remain as it is.

**Follow up:**

* Could you solve the problem in O(1) extra memory space?
* You may not alter the values in the list's nodes, only nodes itself may be changed.

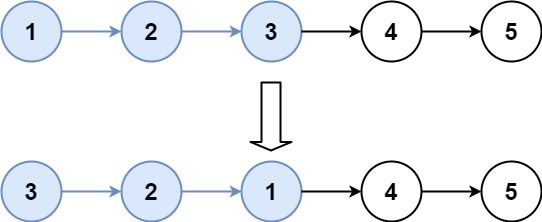
**Example 1:**



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

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

**Example 2:**



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

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

**Example 3:**

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

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

**Example 4:**

**Input:** head = [1], k = 1

**Output:** [1]

**Constraints:**

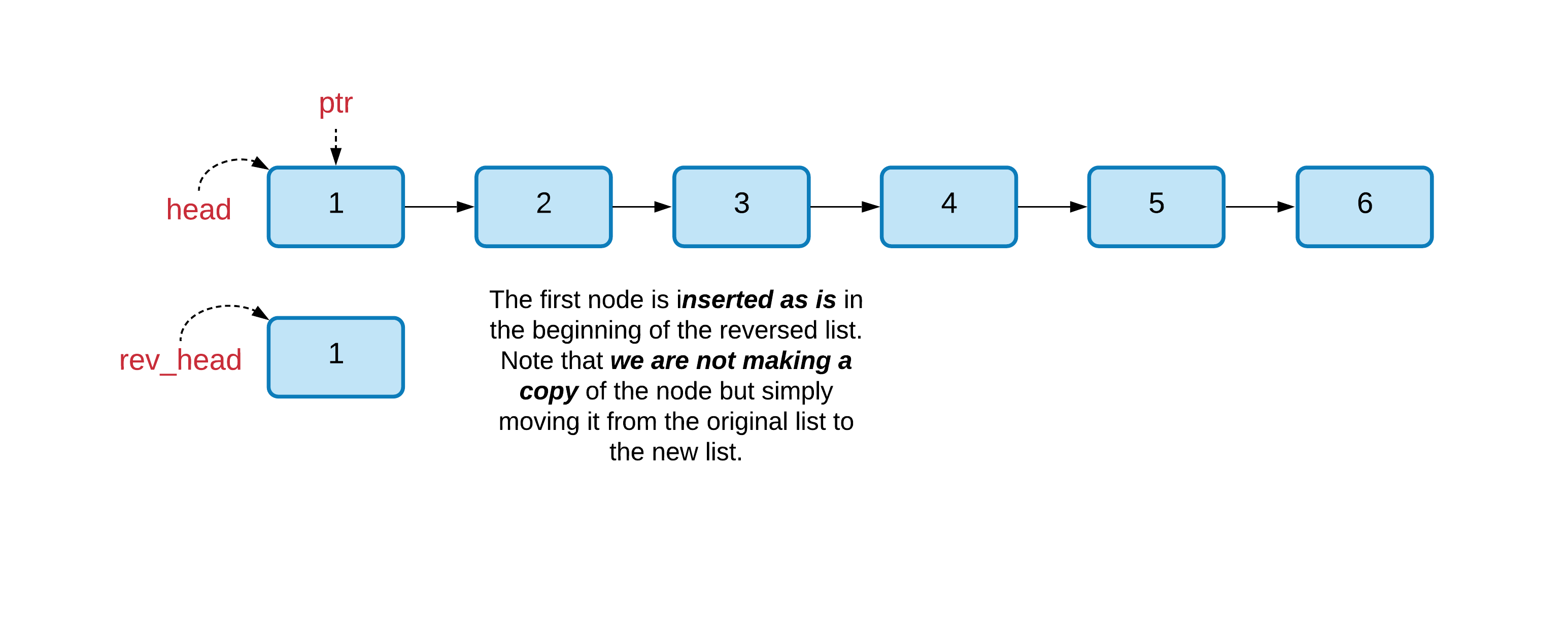
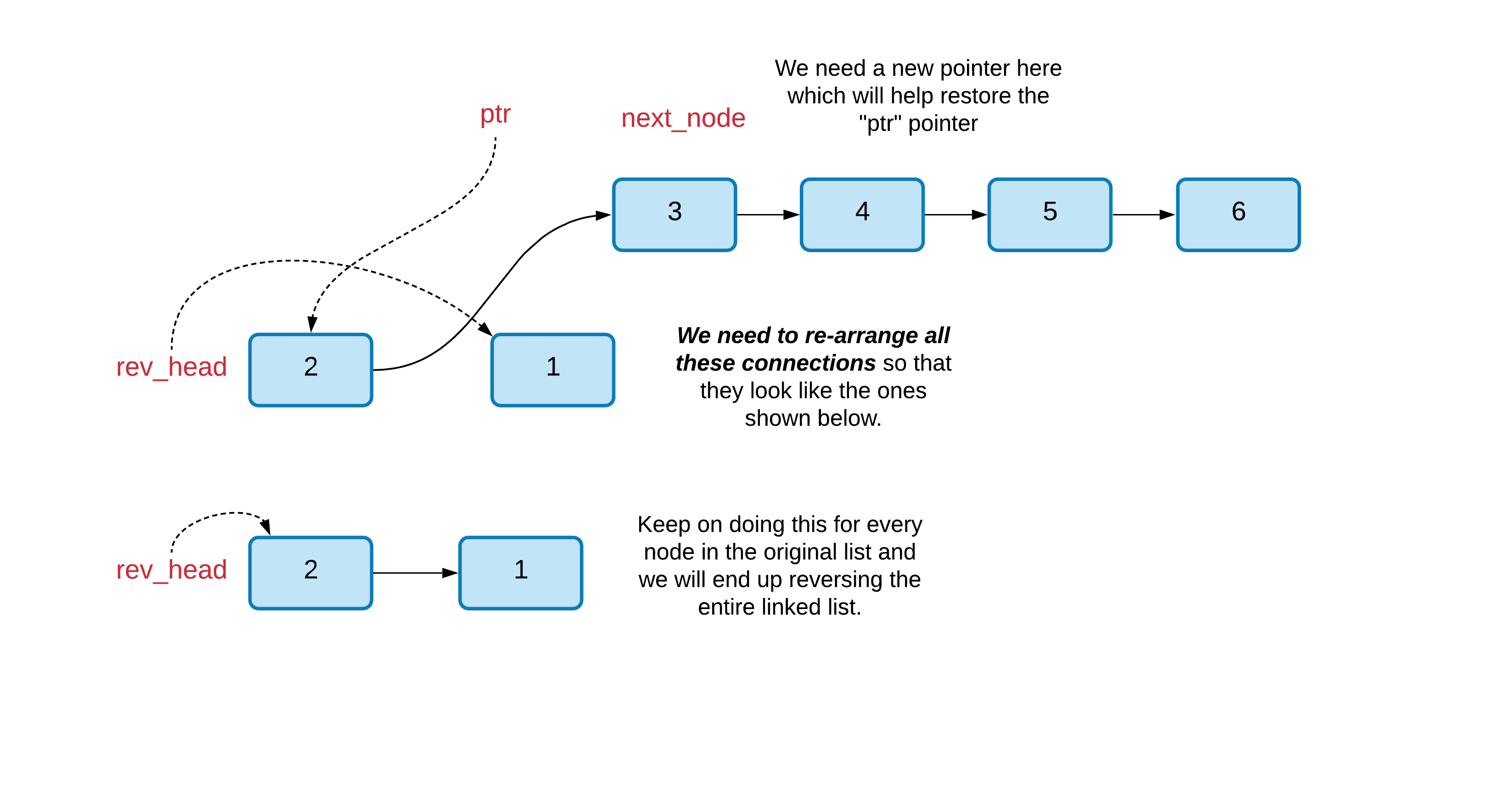
* The number of nodes in the list is in the range sz.
* 1 <= sz <= 5000
* 0 <= Node.val <= 1000
* 1 <= k <= sz

## Solution

The problem statement clearly mentions that we are not to use any additional space for our solution. So naturally, a recursive solution is not acceptable here because of the space utilized by the recursion stack. However, for the sake of completeness, we shall go over the recursive approach first before moving on to the iterative approach. The interviewer may not specify the space constraint initially and so, a recursive solution would be a quick first approach followed by the iterative version.

A Linked list is a recursive structure. A sub-list in itself is a linked list. So, if you think about it, reversing a list consisting of k nodes is simply a linked list reversal algorithm. So, before we look at our actual approaches, we need to know that we will essentially be making use of a linked list reversal function here. There are many ways of reversing a linked list. A lot of programmers like to reverse the links themselves for reversing a linked list. What I personally like to do is to combine linked list traversal with insertion in beginning.

* Say the linked list we need to reverse has the starting node called head.
* Now, we will consider another pointer which will act as the head of the reversed linked list. Let's call this rev\_head.
* We will use a pointer, ptr to traverse the original list.
* For every element, we basically insert it at the beginning of the reverse list which has rev\_head as its head.
* That's it! We keep on moving ptr one step forward and keep inserting the nodes in the beginning of our reverse list and we will end up reversing the entire list.

Now that we have the basic linked list reversal stuff out of the way, we can move on with our actual problem which is to reverse the linked list, k nodes at a time. The basic idea is to make use of our reversal function for a linked list. Usually, we start with the head of the list and keep running the reversal algorithm all the way to the end. However, in this case, we will only process k nodes.

However, the problem statement also mentions that if there are < k nodes left in the linked list, then we don't have to reverse them. This implies that we first need to count k nodes before we get on with our reversal. If at any point, we find that we don't have k nodes, then we don't reverse that portion of the linked list. Right off the bat, this implies at least two traversals of the list overall. One for counting, and the next, for reversals.

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

**Intuition**

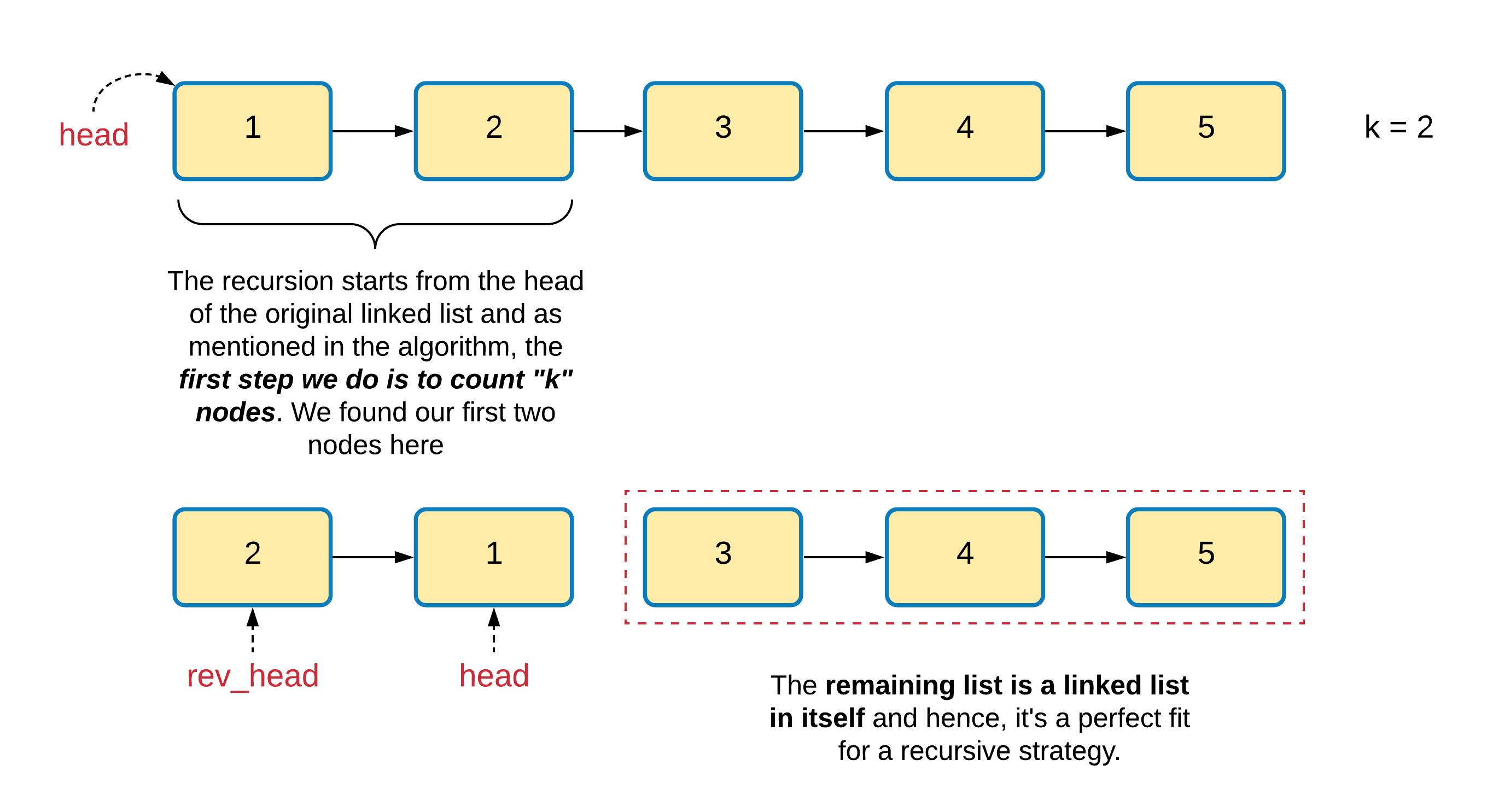
The recursive approach is a natural fit for this problem since the problem asks us to perform a modification operation on a fixed portion of the linked list, one portion at a time. Since a sub-list of a linked list is a linked list in itself, we can make use of recursion to do the heavy lifting for us. All we need to focus here is how we are going to reverse those k nodes. This part is sorted because we already discussed how general linked list reversal works.

We also need to make sure we are hooking up the right connections as recursion backtracks. For e.g. say we are given a linked list 1,2,3,4,5 and we are to reverse two nodes at a time. In the recursive approach, we will first reverse the first two nodes thus getting 2,1. When recursion backtracks, we assume that we will have 4,3,5. Now, we need to ensure that we hookup 1->4 correctly so that the overall list is what we expect.

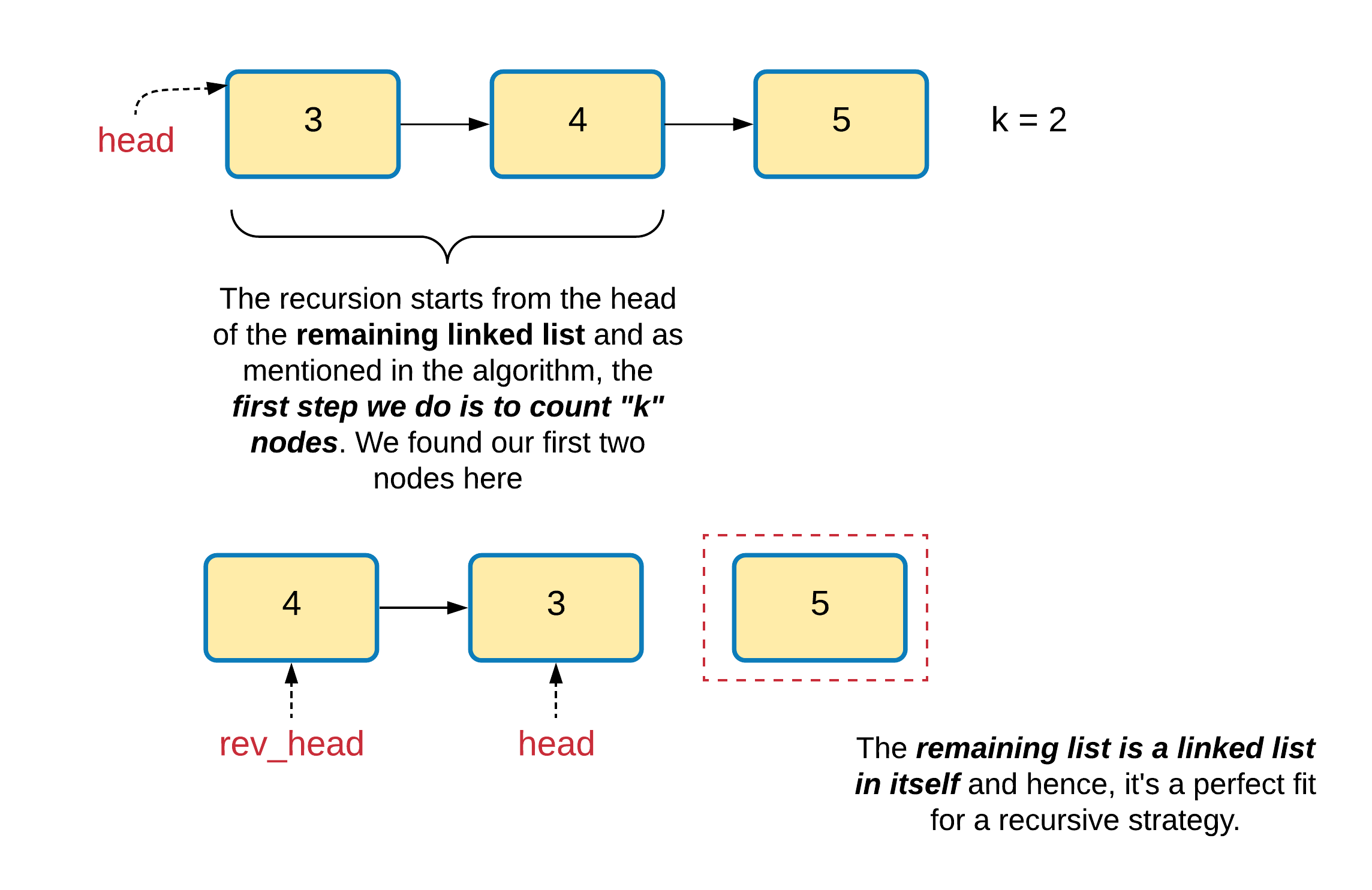
**Algorithm**

1. Assuming we have a reverse() function already defined for a linked list. This function would take the head of the linked list and also an integer value representing k. We don't have to reverse till the end of the linked list. Only k nodes are to be touched at a time.
2. In every recursive call, we first count the number of nodes in the linked list. As soon as the count reaches k, we break.
3. If there are less than k nodes left in the list, we return the head of the list.
4. However, if there are at least k nodes in the list, then we reverse these nodes by calling our reverse() function defined in the first step.
5. Our recursion function needs to return the head of the reversed linked list. This would simply be the k^th*kth* nodes in the list passed to the recursion function because after reversing the first k nodes, the k^th*kth* node will become the new head and so on.
6. So, in every recursive call, we first reverse k nodes, then recurse on the rest of the linked list. When recursion returns, we establish the proper connections.

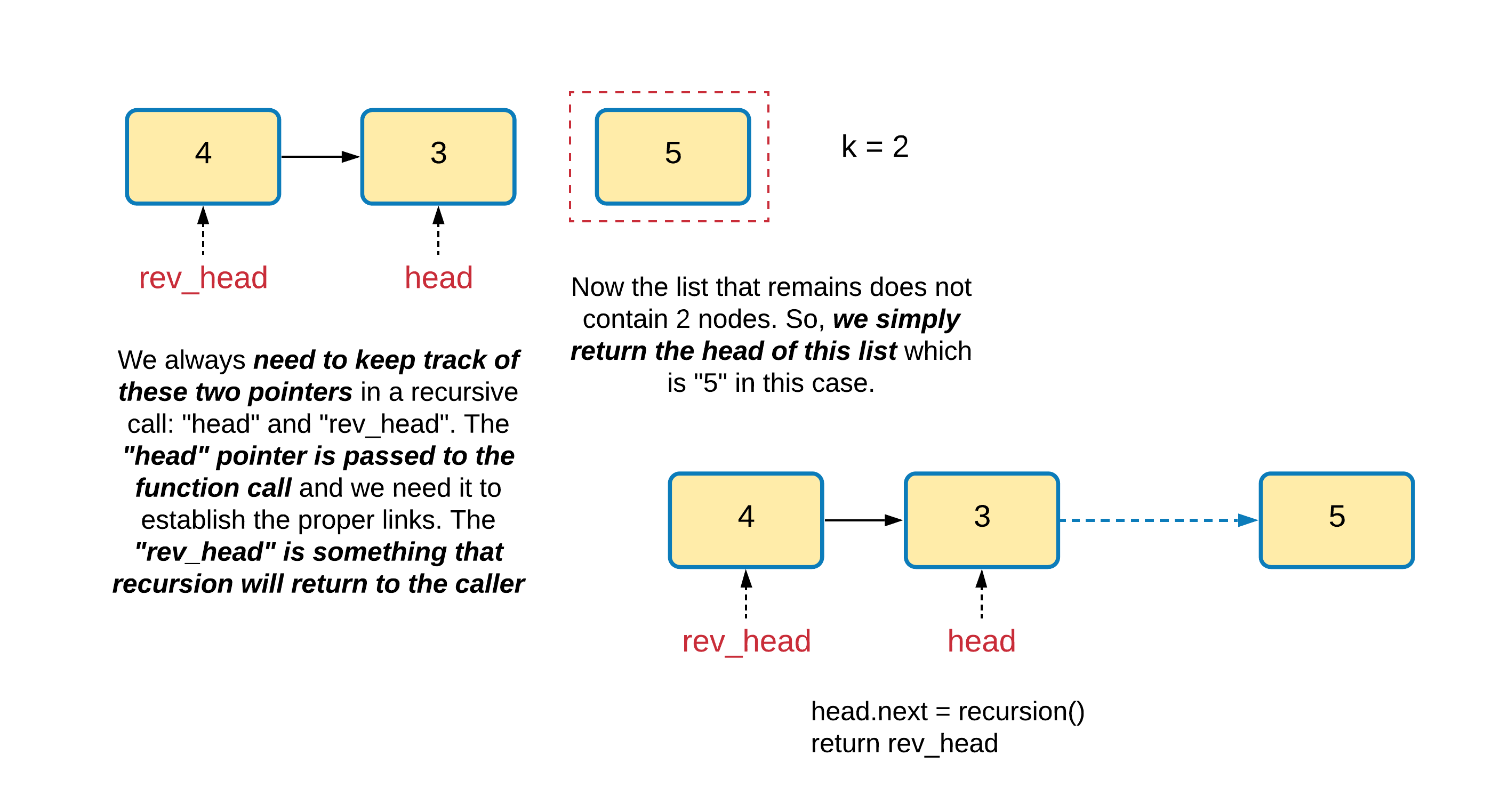
Let's look at a quick example of the algorithm's dry run. So, in the first recursive step, we process the first two nodes of the list and then make a recursive call.



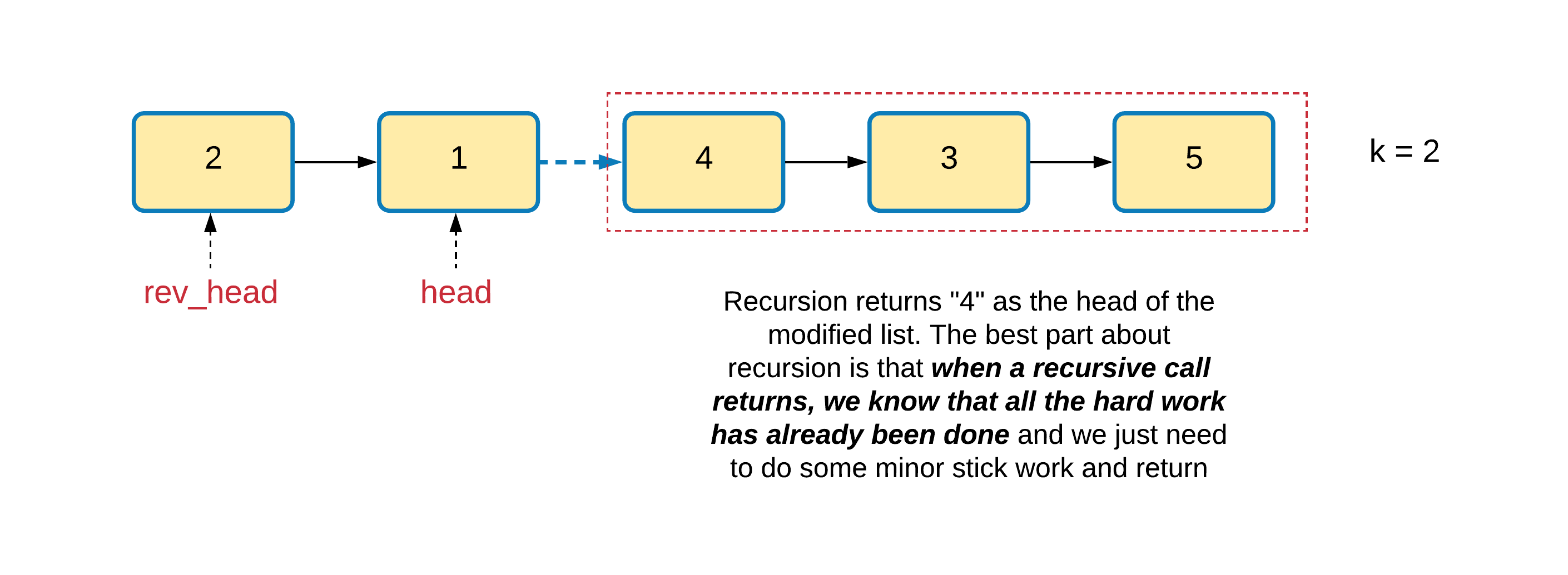
Here again, we process the two nodes and then make the final recursive call for this example linked list.



Now here we don't have enough nodes to reverse. So, in the recursive call we simply return the only remaining node here which is "5". Once that node is returned from the recursive call, we need to establish the proper connections i.e. from 3->5.



Similarly, recursion would return 4 as the new head node of the modified list ahead. We need to establish the connection from 1 to 4 and then return 2 as the head of the modified list.



|  |
| --- |
| /\*\*  \* Definition for singly-linked list.  \* public class ListNode {  \* int val;  \* ListNode next;  \* ListNode(int x) { val = x; }  \* }  \*/  class Solution {    public ListNode reverseLinkedList(ListNode head, int k) {    // Reverse k nodes of the given linked list.  // This function assumes that the list contains  // atleast k nodes.  ListNode new\_head = null;  ListNode ptr = head;    while (k > 0) {    // Keep track of the next node to process in the  // original list  ListNode next\_node = ptr.next;    // Insert the node pointed to by "ptr"  // at the beginning of the reversed list  ptr.next = new\_head;  new\_head = ptr;    // Move on to the next node  ptr = next\_node;    // Decrement the count of nodes to be reversed by 1  k--;  }      // Return the head of the reversed list  return new\_head;    }    public ListNode reverseKGroup(ListNode head, int k) {    int count = 0;  ListNode ptr = head;    // First, see if there are atleast k nodes  // left in the linked list.  while (count < k && ptr != null) {  ptr = ptr.next;  count++;  }      // If we have k nodes, then we reverse them  if (count == k) {    // Reverse the first k nodes of the list and  // get the reversed list's head.  ListNode reversedHead = this.reverseLinkedList(head, k);    // Now recurse on the remaining linked list. Since  // our recursion returns the head of the overall processed  // list, we use that and the "original" head of the "k" nodes  // to re-wire the connections.  head.next = this.reverseKGroup(ptr, k);  return reversedHead;  }    return head;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*) since we process each node exactly twice. Once when we are counting the number of nodes in each recursive call, and then once when we are actually reversing the sub-list. A slightly optimized implementation here could be that we don't count the number of nodes at all and simply reverse k nodes. If at any point we find that we didn't have enough nodes, we can re-reverse the last set of nodes so as to keep the original structure as required by the problem statement. That ways, we can get rid of the extra counting.
* Space Complexity: O(N/k)*O*(*N*/*k*) used up by the recursion stack. The number of recursion calls is determined by both k*k* and N*N*. In every recursive call, we process k*k* nodes and then make a recursive call to process the rest.

#### **Approach 2: Iterative O(1) space**

**Intuition**

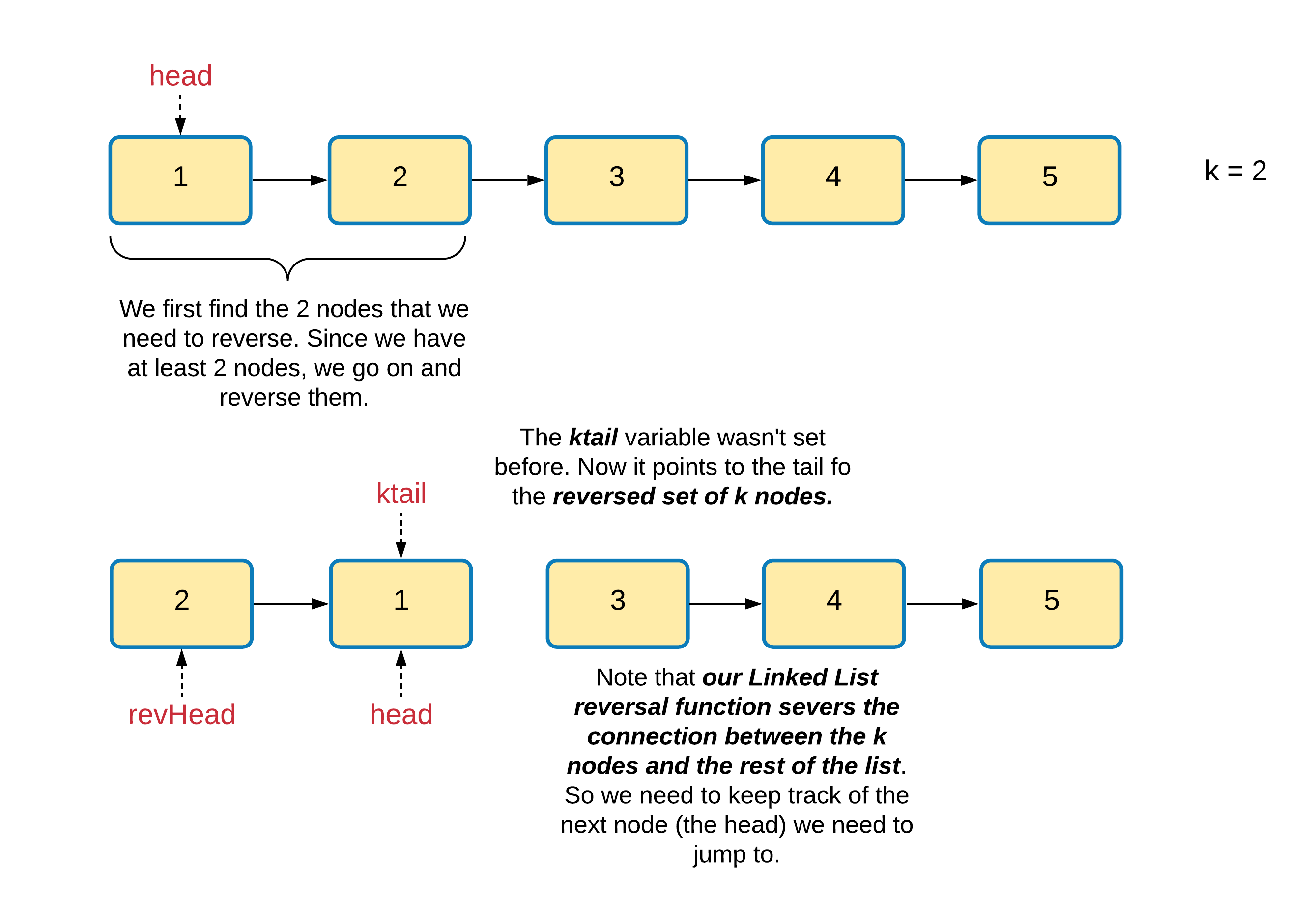
The idea here is the same as before except that we won't be making use of the stack here and rather use a couple additional variables to maintain the proper connections along the way. We still count k nodes at a time. If we find k nodes, then we reverse them.

In addition to the "head" and "rev\_head" variables from before, we need to know the "tail" node of the previous set of k nodes as well. The recursive approach reverses k nodes from left to right, but it establishes the connections from right to left or back to front. In this approach we will be reversing and establishing the connections while going from front to back.

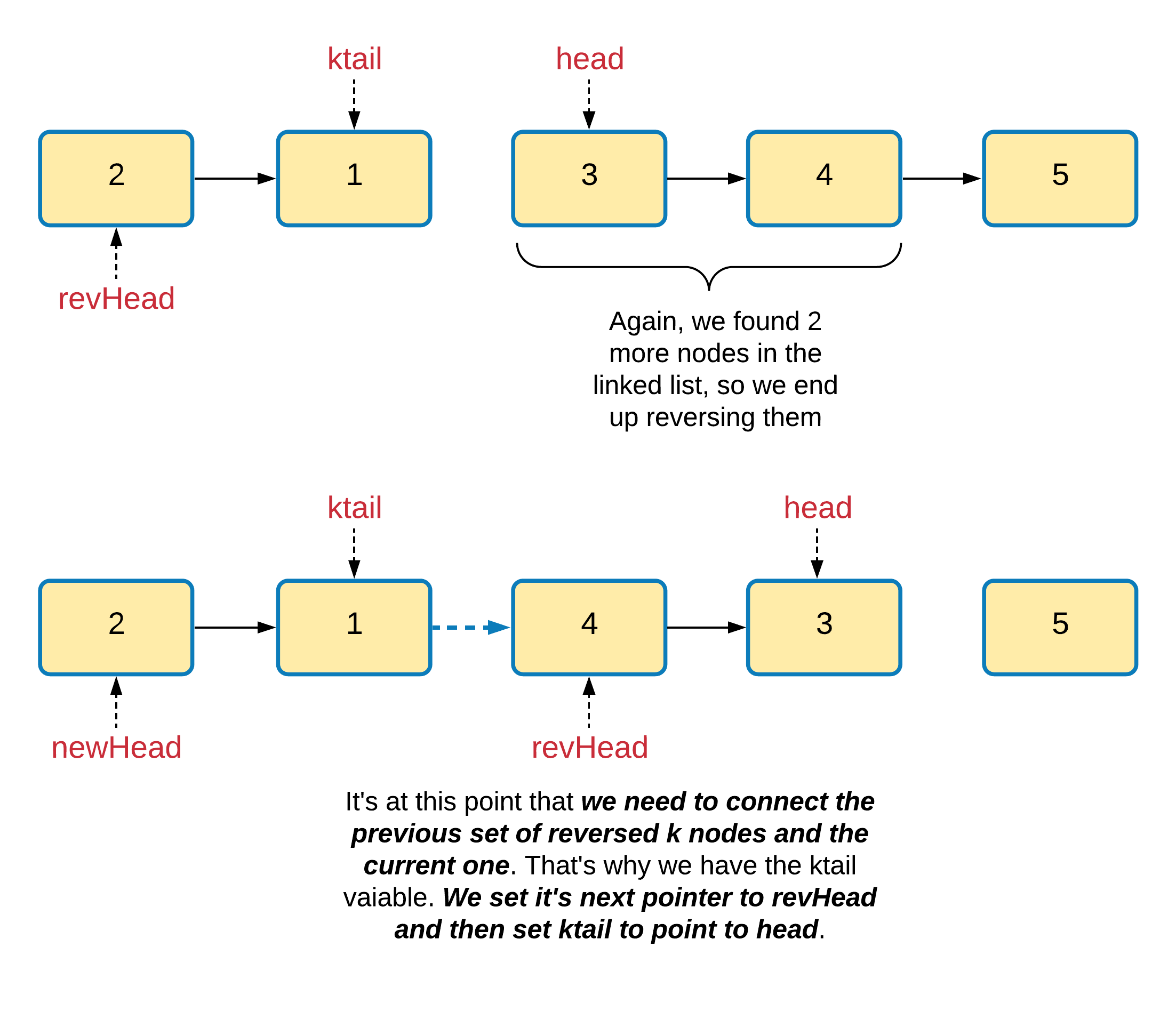
**Algorithm**

1. Assuming we have a reverse() function already defined for a linked list. This function would take the head of the linked list and also an integer value representing k. We don't have to reverse till the end of the linked list. Only k nodes are to be touched at a time.
2. We need to maintain four different variables in this algorithm as we chug along:
   1. head ~ which will always point to the original head of the next set of k nodes.
   2. revHead ~ which is basically the tail node of the original set of k nodes. Hence, this becomes the new head after reversal.
   3. ktail ~ is the tail node of the previous set of k nodes after reversal.
   4. newHead ~ acts as the head of the final list that we need to return as the output. Basically, this is the k^{th}*kth* node from the beginning of the original list.
3. The core algorithm remains the same as before. Given the head, we first count k nodes. If we are able to find at least k nodes, we reverse them and get our revHead.
4. At this point we check if we already have the variable ktail set or not. It won't be set when we reverse the very first set of k nodes. However, if this variable is set, then we attach ktail.next to the revHead. Also, we need to update ktail to point to the tail of the reversed set of k nodes that we just processed. Remember, the head node becomes the new tail and hence, we set ktail = head.
5. We keep doing this until we reach the end of the list or we encounter that there are < k nodes left in the list.

Let's look at the same linked list that we use for a dry run in the first approach. The first step simply assigns all the relevant pointers and reverses the first two nodes.



This step is really important since it highlights the use case of the ktail pointer here.



|  |
| --- |
| class Solution {    public ListNode reverseLinkedList(ListNode head, int k) {    // Reverse k nodes of the given linked list.  // This function assumes that the list contains  // atleast k nodes.  ListNode new\_head = null;  ListNode ptr = head;    while (k > 0) {    // Keep track of the next node to process in the  // original list  ListNode next\_node = ptr.next;    // Insert the node pointed to by "ptr"  // at the beginning of the reversed list  ptr.next = new\_head;  new\_head = ptr;    // Move on to the next node  ptr = next\_node;    // Decrement the count of nodes to be reversed by 1  k--;  }      // Return the head of the reversed list  return new\_head;    }    public ListNode reverseKGroup(ListNode head, int k) {    ListNode ptr = head;  ListNode ktail = null;    // Head of the final, moified linked list  ListNode new\_head = null;    // Keep going until there are nodes in the list  while (ptr != null) {    int count = 0;    // Start counting nodes from the head  ptr = head;    // Find the head of the next k nodes  while (count < k && ptr != null) {  ptr = ptr.next;  count += 1;  }  // If we counted k nodes, reverse them  if (count == k) {    // Reverse k nodes and get the new head  ListNode revHead = this.reverseLinkedList(head, k);    // new\_head is the head of the final linked list  if (new\_head == null)  new\_head = revHead;    // ktail is the tail of the previous block of  // reversed k nodes  if (ktail != null)  ktail.next = revHead;    ktail = head;  head = ptr;  }  }    // attach the final, possibly un-reversed portion  if (ktail != null)  ktail.next = head;    return new\_head == null ? head : new\_head;  }  } |

* Time Complexity: O(N)*O*(*N*) since we process each node exactly twice. Once when we are counting the number of nodes in each recursive call, and then once when we are actually reversing the sub-list.
* Space Complexity: O(1)*O*(1).

**Word Ladder II**

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words such that:

* The first word in the sequence is beginWord.
* The last word in the sequence is endWord.
* Only one letter is different between each adjacent pair of words in the sequence.
* Every word in the sequence is in wordList.

Given two words, beginWord and endWord, and a dictionary wordList, return *all the****shortest transformation sequences****from* beginWord *to* endWord*, or an empty list if no such sequence exists.*

**Example 1:**

**Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

**Output:** [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]

**Example 2:**

**Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

**Output:** []

**Explanation:** The endWord "cog" is not in wordList, therefore no possibletransformation.

**Constraints:**

* 1 <= beginWord.length <= 10
* endWord.length == beginWord.length
* 1 <= wordList.length <= 5000
* wordList[i].length == beginWord.length
* beginWord, endWord, and wordList[i] consist of lowercase English letters.
* beginWord != endWord
* All the strings in wordList are **unique**.

**Cut Off Trees for Golf Event**

You are asked to cut off all the trees in a forest for a golf event. The forest is represented as an m x n matrix. In this matrix:

* 0 means the cell cannot be walked through.
* 1 represents an empty cell that can be walked through.
* A number greater than 1 represents a tree in a cell that can be walked through, and this number is the tree's height.

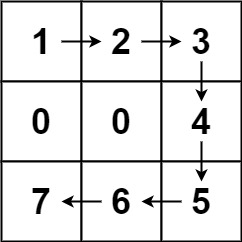
In one step, you can walk in any of the four directions: north, east, south, and west. If you are standing in a cell with a tree, you can choose whether to cut it off.

You must cut off the trees in order from shortest to tallest. When you cut off a tree, the value at its cell becomes 1 (an empty cell).

Starting from the point (0, 0), return *the minimum steps you need to walk to cut off all the trees*. If you cannot cut off all the trees, return -1.

You are guaranteed that no two trees have the same height, and there is at least one tree needs to be cut off.

**Example 1:**

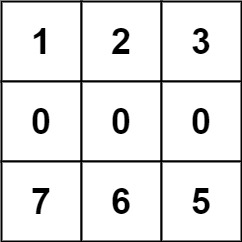


**Input:** forest = [[1,2,3],[0,0,4],[7,6,5]]

**Output:** 6

**Explanation:** Following the path above allows you to cut off the trees from shortest to tallest in 6 steps.

**Example 2:**



**Input:** forest = [[1,2,3],[0,0,0],[7,6,5]]

**Output:** -1

**Explanation:** The trees in the bottom row cannot be accessed as the middle row is blocked.

**Example 3:**

**Input:** forest = [[2,3,4],[0,0,5],[8,7,6]]

**Output:** 6

**Explanation:** You can follow the same path as Example 1 to cut off all the trees.

Note that you can cut off the first tree at (0, 0) before making any steps.

**Constraints:**

* m == forest.length
* n == forest[i].length
* 1 <= m, n <= 50
* 0 <= forest[i][j] <= 109

#### **Approach Framework**

**Explanation**

Starting from (0, 0), for each tree in height order, we will calculate the distance from where we are to the next tree (and move there), adding that distance to the answer.

We frame the problem as providing some distance function dist(forest, sr, sc, tr, tc) that calculates the path distance from source (sr, sc) to target (tr, tc) through obstacles dist[i][j] == 0. (This distance function will return -1 if the path is impossible.)

What follows is code and complexity analysis that is common to all three approaches. After, the algorithms presented in our approaches will focus on only providing our dist function.

|  |
| --- |
| class Solution {  int[] dr = {-1, 1, 0, 0};  int[] dc = {0, 0, -1, 1};  public int cutOffTree(List<List<Integer>> forest) {  List<int[]> trees = new ArrayList();  for (int r = 0; r < forest.size(); ++r) {  for (int c = 0; c < forest.get(0).size(); ++c) {  int v = forest.get(r).get(c);  if (v > 1) trees.add(new int[]{v, r, c});  }  }  Collections.sort(trees, (a, b) -> Integer.compare(a[0], b[0]));  int ans = 0, sr = 0, sc = 0;  for (int[] tree: trees) {  int d = dist(forest, sr, sc, tree[1], tree[2]);  if (d < 0) return -1;  ans += d;  sr = tree[1]; sc = tree[2];  }  return ans;  }  } |

**Complexity Analysis**

All three algorithms have similar worst case complexities, but in practice each successive algorithm presented performs faster on random data.

* Time Complexity: O((RC)^2)*O*((*RC*)2) where there are R*R* rows and C*C* columns in the given forest. We walk to R\*C*R*∗*C* trees, and each walk could spend O(R\*C)*O*(*R*∗*C*) time searching for the tree.
* Space Complexity: O(R\*C)*O*(*R*∗*C*), the maximum size of the data structures used.

#### **Approach #1: BFS [Accepted]**

**Intuition and Algorithm**

We perform a breadth-first-search, processing nodes (grid positions) in a queue. seen keeps track of nodes that have already been added to the queue at some point - those nodes will be already processed or are in the queue awaiting processing.

For each node that is next to be processed, we look at it's neighbors. If they are in the forest (grid), they haven't been enqueued, and they aren't an obstacle, we will enqueue that neighbor.

We also keep a side count of the distance travelled for each node. If the node we are processing is our destination 'target' (tr, tc), we'll return the answer.

|  |
| --- |
| public int bfs(List<List<Integer>> forest, int sr, int sc, int tr, int tc) {  int R = forest.size(), C = forest.get(0).size();  Queue<int[]> queue = new LinkedList();  queue.add(new int[]{sr, sc, 0});  boolean[][] seen = new boolean[R][C];  seen[sr][sc] = true;  while (!queue.isEmpty()) {  int[] cur = queue.poll();  if (cur[0] == tr && cur[1] == tc) return cur[2];  for (int di = 0; di < 4; ++di) {  int r = cur[0] + dr[di];  int c = cur[1] + dc[di];  if (0 <= r && r < R && 0 <= c && c < C &&  !seen[r][c] && forest.get(r).get(c) > 0) {  seen[r][c] = true;  queue.add(new int[]{r, c, cur[2]+1});  }  }  }  return -1;  } |

#### **Approach #2: A\* Search [Accepted]**

**Intuition and Algorithm**

The A\* star algorithm is another path-finding algorithm. For every node at position (r, c), we have some estimated cost node.f = node.g + node.h, where node.g is the actual distance from (sr, sc) to (r, c), and node.h is our heuristic (guess) of the distance from (r, c) to (tr, tc). In this case, our guess will be the taxicab distance, node.h = abs(r-tr) + abs(c-tc).

We keep a priority queue to decide what node to search in (expand) next. We can prove that if we find the target node, we must have travelled the lowest possible distance node.g. By considering the last time where two backwards paths are the same, without loss of generality we could suppose the penultimate square of the two paths are different, and then in this case node.f = node.g + 1, showing the path with less actual distance travelled is expanded first as desired.

It might be useful for solvers familiar with Dijkstra's Algorithm to know that Dijkstra's algorithm is a special case of A\* Search with node.h = 0 always.

|  |
| --- |
| public int cutOffTree(List<List<Integer>> forest, int sr, int sc, int tr, int tc) {  int R = forest.size(), C = forest.get(0).size();  PriorityQueue<int[]> heap = new PriorityQueue<int[]>(  (a, b) -> Integer.compare(a[0], b[0]));  heap.offer(new int[]{0, 0, sr, sc});  HashMap<Integer, Integer> cost = new HashMap();  cost.put(sr \* C + sc, 0);  while (!heap.isEmpty()) {  int[] cur = heap.poll();  int g = cur[1], r = cur[2], c = cur[3];  if (r == tr && c == tc) return g;  for (int di = 0; di < 4; ++di) {  int nr = r + dr[di], nc = c + dc[di];  if (0 <= nr && nr < R && 0 <= nc && nc < C && forest.get(nr).get(nc) > 0) {  int ncost = g + 1 + Math.abs(nr-tr) + Math.abs(nc-tr);  if (ncost < cost.getOrDefault(nr \* C + nc, 9999)) {  cost.put(nr \* C + nc, ncost);  heap.offer(new int[]{ncost, g+1, nr, nc});  }  }  }  }  return -1;  } |

#### **Approach #3: Hadlock's Algorithm [Accepted]**

**Intuition**

Without any obstacles, the distance from source = (sr, sc) to target = (tr, tc) is simply taxi(source, target) = abs(sr-tr) + abs(sc-tc). This represents a sort of minimum distance that must be travelled. Whenever we walk "away" from the target, we increase this minimum by 2, as we stepped 1 move, plus the taxicab distance from our new location has increased by one.

Let's call such a move that walks away from the target a detour. It can be proven that the distance from source to target is simply taxi(source, target) + 2 \* detours, where detours is the smallest number of detours in any path from source to target.

**Algorithm**

With respect to a source and target, call the detour number of a square to be the lowest number of detours possible in any path from source to that square. (Here, detours are defined with respect to target - the number of away steps from that target.)

We will perform a priority-first-search in order of detour number. If the target is found, it was found with the lowest detour number and therefore the lowest corresponding distance. This motivates using processed, keeping track of when nodes are expanded, not visited - nodes could potentially be visited twice.

As each neighboring node can only have the same detour number or a detour number one higher, we will only consider at most 2 priority classes at a time. Thus, we can use a deque (double ended queue) to perform this implementation. We will place nodes with the same detour number to be expanded first, and nodes with a detour number one higher to be expanded after all nodes with the current number are done.

|  |
| --- |
| public int hadlocks(List<List<Integer>> forest, int sr, int sc, int tr, int tc) {  int R = forest.size(), C = forest.get(0).size();  Set<Integer> processed = new HashSet();  Deque<int[]> deque = new ArrayDeque();  deque.offerFirst(new int[]{0, sr, sc});  while (!deque.isEmpty()) {  int[] cur = deque.pollFirst();  int detours = cur[0], r = cur[1], c = cur[2];  if (!processed.contains(r\*C + c)) {  processed.add(r\*C + c);  if (r == tr && c == tc) {  return Math.abs(sr-tr) + Math.abs(sc-tc) + 2 \* detours;  }  for (int di = 0; di < 4; ++di) {  int nr = r + dr[di];  int nc = c + dc[di];  boolean closer;  if (di <= 1) closer = di == 0 ? r > tr : r < tr;  else closer = di == 2 ? c > tc : c < tc;  if (0 <= nr && nr < R && 0 <= nc && nc < C && forest.get(nr).get(nc) > 0) {  if (closer) deque.offerFirst(new int[]{detours, nr, nc});  else deque.offerLast(new int[]{detours+1, nr, nc});  }  }  }  }  return -1;  } |

**Maximum Frequency Stack**

Implement FreqStack, a class which simulates the operation of a stack-like data structure.

FreqStack has two functions:

* push(int x), which pushes an integer x onto the stack.
* pop(), which **removes** and returns the most frequent element in the stack.
  + If there is a tie for most frequent element, the element closest to the top of the stack is removed and returned.

**Example 1:**

**Input:**

["FreqStack","push","push","push","push","push","push","pop","pop","pop","pop"],

[[],[5],[7],[5],[7],[4],[5],[],[],[],[]]

**Output:** [null,null,null,null,null,null,null,5,7,5,4]

**Explanation**:

After making six .push operations, the stack is [5,7,5,7,4,5] from bottom to top. Then:

pop() -> returns 5, as 5 is the most frequent.

The stack becomes [5,7,5,7,4].

pop() -> returns 7, as 5 and 7 is the most frequent, but 7 is closest to the top.

The stack becomes [5,7,5,4].

pop() -> returns 5.

The stack becomes [5,7,4].

pop() -> returns 4.

The stack becomes [5,7].

**Note:**

* Calls to FreqStack.push(int x) will be such that 0 <= x <= 10^9.
* It is guaranteed that FreqStack.pop() won't be called if the stack has zero elements.
* The total number of FreqStack.push calls will not exceed 10000 in a single test case.
* The total number of FreqStack.pop calls will not exceed 10000 in a single test case.
* The total number of FreqStack.push and FreqStack.pop calls will not exceed 150000 across all test cases.

## Solution

#### **Approach 1: Stack of Stacks**

**Intuition**

Evidently, we care about the frequency of an element. Let freq be a Map from x*x* to the number of occurrences of x*x*.

Also, we (probably) care about maxfreq, the current maximum frequency of any element in the stack. This is clear because we must pop the element with the maximum frequency.

The main question then becomes: among elements with the same (maximum) frequency, how do we know which element is most recent? We can use a stack to query this information: the top of the stack is the most recent.

To this end, let group be a map from frequency to a stack of elements with that frequency. We now have all the required components to implement FreqStack.

**Algorithm**

Actually, as an implementation level detail, if x has frequency f, then we'll have x in all group[i] (i <= f), not just the top. This is because each group[i] will store information related to the ith copy of x.

Afterwards, our goal is just to maintain freq, group, and maxfreq as described above.

|  |
| --- |
| class FreqStack {  Map<Integer, Integer> freq;  Map<Integer, Stack<Integer>> group;  int maxfreq;  public FreqStack() {  freq = new HashMap();  group = new HashMap();  maxfreq = 0;  }  public void push(int x) {  int f = freq.getOrDefault(x, 0) + 1;  freq.put(x, f);  if (f > maxfreq)  maxfreq = f;  group.computeIfAbsent(f, z-> new Stack()).push(x);  }  public int pop() {  int x = group.get(maxfreq).pop();  freq.put(x, freq.get(x) - 1);  if (group.get(maxfreq).size() == 0)  maxfreq--;  return x;  }  } |

**Complexity Analysis**

* Time Complexity: O(1)*O*(1) for both push and pop operations.
* Space Complexity: O(N)*O*(*N*), where N is the number of elements in the FreqStack.

**Partition Labels**

A string S of lowercase English letters is given. We want to partition this string into as many parts as possible so that each letter appears in at most one part, and return a list of integers representing the size of these parts.

**Example 1:**

**Input:** S = "ababcbacadefegdehijhklij"

**Output:** [9,7,8]

**Explanation:**

The partition is "ababcbaca", "defegde", "hijhklij".

This is a partition so that each letter appears in at most one part.

A partition like "ababcbacadefegde", "hijhklij" is incorrect, because it splits S into less parts.

**Note:**

* S will have length in range [1, 500].
* S will consist of lowercase English letters ('a' to 'z') only.

   Hide Hint #1

Try to greedily choose the smallest partition that includes the first letter. If you have something like "abaccbdeffed", then you might need to add b. You can use an map like "last['b'] = 5" to help you expand the width of your partition.

#### **Approach 1: Greedy**

**Intuition**

Let's try to repeatedly choose the smallest left-justified partition. Consider the first label, say it's 'a'. The first partition must include it, and also the last occurrence of 'a'. However, between those two occurrences of 'a', there could be other labels that make the minimum size of this partition bigger. For example, in "abccaddbeffe", the minimum first partition is "abccaddb". This gives us the idea for the algorithm: For each letter encountered, process the last occurrence of that letter, extending the current partition [anchor, j] appropriately.

**Algorithm**

We need an array last[char] -> index of S where char occurs last. Then, let anchor and j be the start and end of the current partition. If we are at a label that occurs last at some index after j, we'll extend the partition j = last[c]. If we are at the end of the partition (i == j) then we'll append a partition size to our answer, and set the start of our new partition to i+1.

|  |
| --- |
| class Solution {  public List<Integer> partitionLabels(String S) {  int[] last = new int[26];  for (int i = 0; i < S.length(); ++i)  last[S.charAt(i) - 'a'] = i;    int j = 0, anchor = 0;  List<Integer> ans = new ArrayList();  for (int i = 0; i < S.length(); ++i) {  j = Math.max(j, last[S.charAt(i) - 'a']);  if (i == j) {  ans.add(i - anchor + 1);  anchor = i + 1;  }  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of S*S*.
* Space Complexity: O(1)*O*(1) to keep data structure last of not more than 26 characters.

**Prison Cells After N Days**

There are 8 prison cells in a row and each cell is either occupied or vacant.

Each day, whether the cell is occupied or vacant changes according to the following rules:

* If a cell has two adjacent neighbors that are both occupied or both vacant, then the cell becomes occupied.
* Otherwise, it becomes vacant.

**Note** that because the prison is a row, the first and the last cells in the row can't have two adjacent neighbors.

You are given an integer array cells where cells[i] == 1 if the ith cell is occupied and cells[i] == 0 if the ith cell is vacant, and you are given an integer n.

Return the state of the prison after n days (i.e., n such changes described above).

**Example 1:**

**Input:** cells = [0,1,0,1,1,0,0,1], n = 7

**Output:** [0,0,1,1,0,0,0,0]

**Explanation:** The following table summarizes the state of the prison on each day:

Day 0: [0, 1, 0, 1, 1, 0, 0, 1]

Day 1: [0, 1, 1, 0, 0, 0, 0, 0]

Day 2: [0, 0, 0, 0, 1, 1, 1, 0]

Day 3: [0, 1, 1, 0, 0, 1, 0, 0]

Day 4: [0, 0, 0, 0, 0, 1, 0, 0]

Day 5: [0, 1, 1, 1, 0, 1, 0, 0]

Day 6: [0, 0, 1, 0, 1, 1, 0, 0]

Day 7: [0, 0, 1, 1, 0, 0, 0, 0]

**Example 2:**

**Input:** cells = [1,0,0,1,0,0,1,0], n = 1000000000

**Output:** [0,0,1,1,1,1,1,0]

**Constraints:**

* cells.length == 8
* cells[i] is either 0 or 1.
* 1 <= n <= 109

## Solution

#### **Overview**

First of all, one can consider this problem as a simplified version of the [Game of Life](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life) invented by the British mathematician John Horton Conway in 1970.

By simplification, this problem is played on one dimensional array (compared to 2D in Game of Life), and it has less rules.

Due to the nature of game, one of the most intuitive solutions to solve this problem is playing the game, i.e. we can simply run the **simulation**.

Starting from the initial state of the prison cells, we could evolve the states following the rules defined in the problem step by step.

In the following sections, we will give some approaches on how to run the simulation efficiently.

#### **Approach 1: Simulation with Fast Forwarding**

**Intuition**

One important observation from the Game of Life is that we would encounter some already-seen state over the time, simply due to the fact that there are limited number of states.

The above observation applies to our problem here as well. Given K*K* number of cells, there could be at most 2^K2*K* possible states. If the number of steps is larger than all possible states (i.e. N \gt 2^K*N*>2*K*), we are destined to repeat ourselves sooner or later.

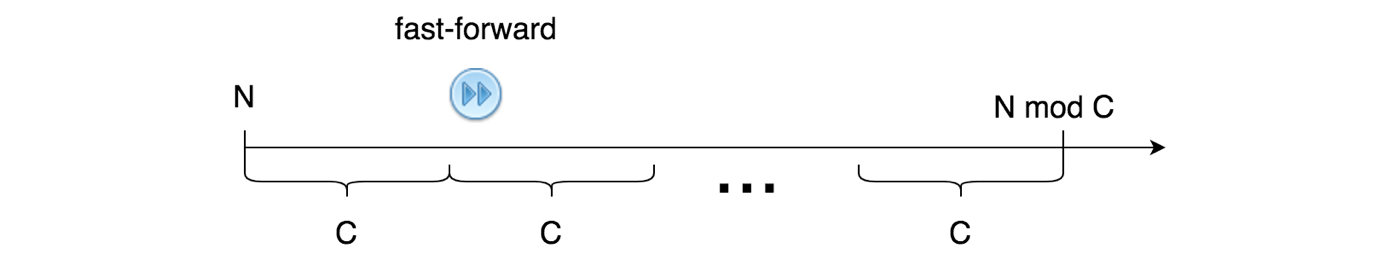
In fact, we would encounter the repetitive states **sooner** than the theoretical boundary we estimated above. For instance, with the initial state of [1,0,0,0,1,0,0,1], just after 15 steps, we would encounter a previously seen state. Once we encounter a state seen before, the history would then repeat itself again and again, assuming that time is infinite.

All states between two repetitive states form a cycle, which would repeat itself over the time. Therefore, based on this observation, we could **fast-forward** the simulation rather than going step by step, once we encounter any repetitive state.

**Algorithm**

Here is the overall idea to implement our fast-forward strategy.

* First of all, we record the state at each step, with the index of the current step, i.e. state -> step\_index.
* Once we discover a repetitive state, we can then determine the **length** (denoted as C*C*) of the cycle, with the help of hashmap that we recorded.
* Starting from this repetitive state, the prison cells would play out the states within the cycle over and over, until we run out of steps.
* In other words, if the remaining steps is N*N*, at least we could **fast-forward** to the step of N \mod C*N*mod*C*.
* And then from the step of N \mod C*N*mod*C*, we continue the simulation step by step.



Note: we only need to do the fast-forward once, if there is any.

Here are some sample implementations based on the above idea.

|  |
| --- |
| class Solution {  protected int cellsToBitmap(int[] cells) {  int stateBitmap = 0x0;  for (int cell : cells) {  stateBitmap <<= 1;  stateBitmap = (stateBitmap | cell);  }  return stateBitmap;  }  protected int[] nextDay(int[] cells) {  int[] newCells = new int[cells.length];  newCells[0] = 0;  for (int i = 1; i < cells.length - 1; i++)  newCells[i] = (cells[i - 1] == cells[i + 1]) ? 1 : 0;  newCells[cells.length - 1] = 0;  return newCells;  }  public int[] prisonAfterNDays(int[] cells, int N) {  HashMap<Integer, Integer> seen = new HashMap<>();  boolean isFastForwarded = false;  // step 1). run the simulation with hashmap  while (N > 0) {  if (!isFastForwarded) {  int stateBitmap = this.cellsToBitmap(cells);  if (seen.containsKey(stateBitmap)) {  // the length of the cycle is seen[state\_key] - N  N %= seen.get(stateBitmap) - N;  isFastForwarded = true;  } else  seen.put(stateBitmap, N);  }  // check if there is still some steps remained,  // with or without the fast-forwarding.  if (N > 0) {  N -= 1;  cells = this.nextDay(cells);  }  }  return cells;  }  } |

**Complexity Analysis**

Let K*K* be the number of cells, and N*N* be the number of steps.

* Time Complexity: \mathcal{O}\big(K \cdot \min(N, 2^K)\big)O(*K*⋅min(*N*,2*K*))
  + As we discussed before, at most we could have 2^K2*K* possible states. While we run the simulation with N*N* steps, we might need to run \min(N, 2^K)min(*N*,2*K*) steps without fast-forwarding in the worst case.
  + For each simulation step, it takes \mathcal{O}(K)O(*K*) time to process and evolve the state of cells.
  + Hence, the overall time complexity of the algorithm is \mathcal{O}\big(K \cdot \min(N, 2^K)\big)O(*K*⋅min(*N*,2*K*)).
* Space Complexity:
  + The main memory consumption of the algorithm is the hashmap that we used to keep track of the states of the cells. The maximal number of entries in the hashmap would be 2^K2*K* as we discussed before.
  + In the Java implementation, we encode the state as a single integer value. Therefore, its space complexity would be \mathcal{O}(2^K)O(2*K*), assuming that K*K* does not exceed 32 so that a state can fit into a single integer number.
  + In the Python implementation, we keep the states of cells as they are in the hashmap. As a result, for each entry, it takes \mathcal{O}(K)O(*K*) space. In total, its space complexity becomes \mathcal{O}(K \cdot 2^K)O(*K*⋅2*K*).

#### **Approach 2: Simulation with Bitmap**

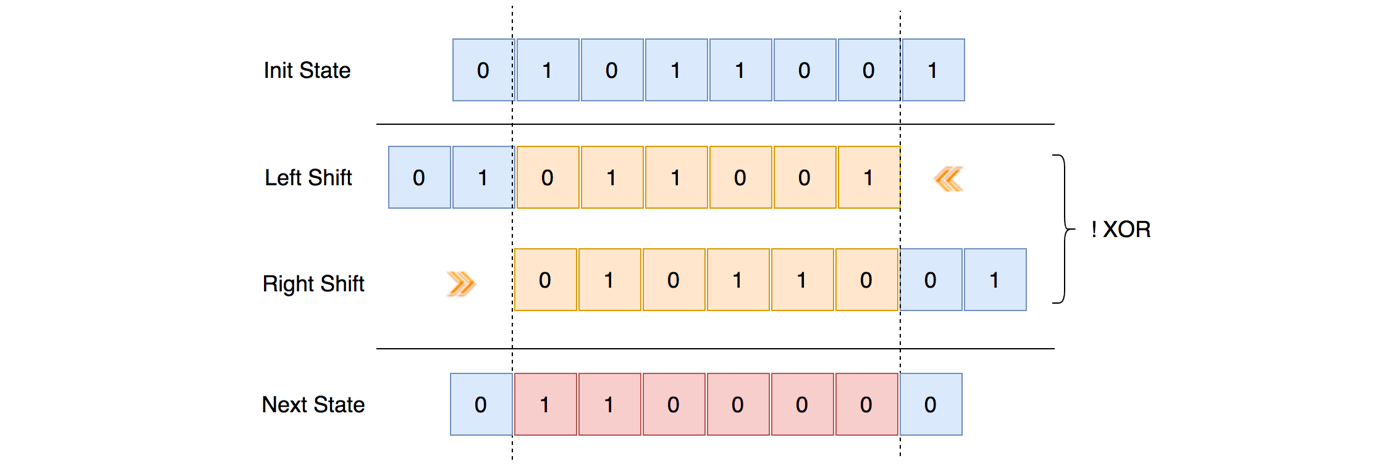
**Intuition**

In the above approach, we implemented the function nextDay(state), which runs an **iteration** to calculate the next state of cells, given the current state.

Given that we have already encoded the state as a bitmap in the Java implementation of the previous approach, a more efficient way to implement the nextDay function would be to apply the **bit operations** (e.g AND, OR, XOR etc.).

The next state of a cell depends on its left and right neighbors. To align the states of its neighbors, we could make a left and a right shift respectively on the bitmap. Upon the shifted bitmaps, we then apply the XOR and NOT operations sequentially, which would lead to the next state of the cell.

Here we show how it works with a concrete example.



Note that, the head and tail cells are particular, which would remain vacant once we start the simulation. Therefore, we should reset the head and tail bits by applying the bit AND operation with the **bitmask** of 01111110 (i.e. 0x7e).

**Algorithm**

We could reuse the bulk of the previous implementations, and simply rewrite the nextDay function with the bit operations as we discussed.

Additionally, at the end of the simulation, we should decode the states of the cells from the final bitmap.

|  |
| --- |
| class Solution {  public int[] prisonAfterNDays(int[] cells, int N) {  HashMap<Integer, Integer> seen = new HashMap<>();  boolean isFastForwarded = false;  // step 1). convert the cells to bitmap  int stateBitmap = 0x0;  for (int cell : cells) {  stateBitmap <<= 1;  stateBitmap = (stateBitmap | cell);  }  // step 2). run the simulation with hashmap  while (N > 0) {  if (!isFastForwarded) {  if (seen.containsKey(stateBitmap)) {  // the length of the cycle is seen[state\_key] - N  N %= seen.get(stateBitmap) - N;  isFastForwarded = true;  } else  seen.put(stateBitmap, N);  }  // check if there is still some steps remained,  // with or without the fast forwarding.  if (N > 0) {  N -= 1;  stateBitmap = this.nextDay(stateBitmap);  }  }  // step 3). convert the bitmap back to the state cells  int ret[] = new int[cells.length];  for (int i = cells.length - 1; i >= 0; i--) {  ret[i] = (stateBitmap & 0x1);  stateBitmap = stateBitmap >> 1;  }  return ret;  }  protected int nextDay(int stateBitmap) {  stateBitmap = ~(stateBitmap << 1) ^ (stateBitmap >> 1);  // set the head and tail to zero  stateBitmap = stateBitmap & 0x7e;  return stateBitmap;  }  } |

**Complexity Analysis**

Let K*K* be the number of cells, and N*N* be the number of steps.

* Time Complexity: \mathcal{O}\big(\min(N, 2^K)\big)O(min(*N*,2*K*)) assuming that K*K* does not exceed 32.
  + As we discussed before, at most we could have 2^K2*K* possible states. While we run the simulation, we need to run \min(N, 2^K)min(*N*,2*K*) steps without fast-forwarding in the worst case.
  + For each simulation step, it takes a constant \mathcal{O}(1)O(1) time to process and evolve the states of cells, since we applied the bit operations rather than iteration.
  + Hence, the overall time complexity of the algorithm is \mathcal{O}\big(\min(N, 2^K)\big)O(min(*N*,2*K*)).
* Space Complexity: \mathcal{O}(2^K)O(2*K*)
  + The main memory consumption of the algorithm is the hashmap that we used to keep track of the states of the cells. The maximal number of entries in the hashmap would be 2^K2*K* as we discussed before.
  + This time we adopted the bitmap for both Java and Python implementation, so that each state consumes a constant \mathcal{O}(1)O(1) space.
  + To sum up, the overall space complexity of the algorithm is \mathcal{O}(2^K)O(2*K*).