LeetCode Problems

Kevin Sony

2025

This is a collection of my notes on particularly difficult problems, and how I arrived at the solution.

Median of Two Sorted Arrays

Given two sorted arrays x and y of size m and n respectively, return the median of the two sorted arrays.

Solution

The easiest way to solve this problem is by combining the arrays, sorting it, then finding the median.

With the knowledge that the two arrays are already sorted though, the sorting method that can be employed is the final step of the merge algorithm, the merging. Keep track of two variables i_1 and i_2, which index x and y respectively. When

$$i_1 + i_2 = \frac{m+n}{2}$$

we have the median. When implementing, in the case $m+n$ is odd, the median is simply whatever value that should be located at the combined index, which is when

$$i_1 + i_2 = \lfloor \frac{m+n}{2} \rfloor$$

In the case $m+n$ is even, the median is the average of the lower and higher median, which is the two values located at

$$i_1 + i_2 = \lfloor \frac{m+n}{2} \rfloor - 1$$
$$i_1 + i_2 = \lfloor \frac{m+n}{2} \rfloor$$

This is what is implemented.

The best solution is to implement a form of binary search.

Zigzag Conversion

The link to the problem is here.

To solve this problem, first transform each index of the original string into a 2D point, then transform the 2D point into another index for the new string. Let s be the length of the string, let n equal the number of rows, and let $m = n + (n - 2)$ represent the number of elements in each "group" before going back to the starting position. The number of "groups" is $\lceil \frac{s}{m} \rceil$, and the amount of elements in the last "incomplete" group is $s \bmod m$.

For the 2D point (x, y), let x represent the column offset, and let y represent the row offset (so $(0, 1)$ is 1 point down from the first element, while $(3, 0)$ is 3 points to the right of the first element). We

We start off with the first index i_1. The "group" it belongs to is $a = \frac{i_1}{m}$, and the corresponding starting point for the 2D representation is $(a(n-1), 0)$. The offset is $b = i \bmod m$. If $b < n$, then add $(0, b)$ to the starting point. Otherwise, add $(0, n-1) + (b-(n-1), (n-1)-b)$, which reduces to $(b-(n-1), 2(n-1)-b)$

From the 2D representation, find the index of the new string, i_2 the 2D point corresponds. The simplest way to implement this would be to put all the 2D points into a data structure, then for each point, count the number of points "smaller than" it, where (x_1, y_1) is "smaller than" (x_2, y_2) if $(x_1, y_1) \preceq (x_2, y_2)$ and $(x_1, y_1) \neq (x_2, y_2)$. This number is i_2.

This requires all the points to have been converted to a corresponding 2D point, meaning all indices must be converted first. This isn't too bad of a requirement, but still something to keep in mind.

Keep an array of the number of points in each "y-value"/row. In determining i_2, sum up all the elements in the array less than the corresponding y-value. Then determine the number of elements with the same y-value but a smaller x-value. This part is much easier, since one only needs to calculate the group they are in, then which offset it is.

Along the same row, in the case where $y = 0$ or $y = n - 1$, the number of points that appears before the corresponding 2D point is a. Otherwise, the number of points that appears in the same row is either $2a$ or $2a + 1$, if $b < n$ or not respectively. Adding it to the number of elements previously, one gets the total number of points that appear before, which is i_2.

Reverse Nodes in k-Group

Given a linked list, for every k elements in the list, reverse it, and if the list doesn't have a multiple of k elements, leave the ends as is. The easiest way to do this is to apply a more general reversed linked list for each of the "groups", then attach the groups back together.

Specifically, split the list into $\lceil n/k \rceil$ groups, the first $\lfloor n/k \rfloor$ groups having k elements, and the last one being a remainder. Then reverse the linked list in the former groups. Finally, join them back together.

Reversing a linked list can be done in $O(m)$ time.

$$A \to B \to C \to D \to E$$
$$E \to D \to C \to B \to A$$

The trick is to note is that this linked

$$E \to D \to C \to B \to A$$

and this one

$$A \leftarrow B \leftarrow C \leftarrow D \leftarrow E$$

are equivalent. The problem then becomes making the child node point to the parent, which is not that difficult..

Sudoku Solver

This can be solved using recursion. Find an empty cell and find a valid value for this cell. If there is a valid value, fill it with that value and go to the next empty cell to repeat this process. If there isn't a valid value, then backtrack.

This problem can be optimized by limiting the number of valid values at the beginning, resulting in less backtracks in general since fewer cells would have to be checked. The number of valid values can be limited by ensuring proper constraints can allow one to fill out as much of the board as possible without resorting to backtracking.

First Missing Positive

Given an array of integers without repeat elements, find the first positive number missing from this array.

The simplest way to solve this problem is to first set all negative values of the array to $n+2$, where n is the size of the array, then sort the array with a sorting algorithm like quicksort, allowing for the problem to be solved in $O(n \log n)$ time complexity and $O(1)$ space complexity.

A better way to decrease time complexity is to create a new boolean array of size n, initialized to all false, then loop through the original array, setting the indices in the boolean array to be true if it is an element in the original array. Then by looping through the boolean array again, wherever the first false is encountered is the first missing positive index not found in the original array. This approach reduces the time complexity to $O(n)$, but increases space complexity to $O(n)$.

The optimal approach is to use a variation of cycle sort. The way cycle sort works is that for a permutation of an array $[1...n]$, represented by $[\sigma(1)...\sigma(n)]$, one can sort this array in $O(n)$ time. This works by a basic fact from group theory, that a permutation σ can be broken up into disjoint cycles. Furthermore, the order of each cycle is less than n, meaning if the cycle is applied to itself for some number less than n times, then all the elements in this cycle returns to its original position. However, there is no easy way of keeping track of all the elements in the cycle. Fortunately, group theory also tells us that cycles can also be written as a product of transpositions.

With all of this, all that is needed to implemented cycle sort is to loop through the array $[\sigma(1)...\sigma(n)]$ once. At each index i, there is a corresponding element $\sigma(i) = j$ such that $1 \leq j \leq n$. Thus swap the contents of indices i and j so that $\sigma(j) = j$. The element in index i is the old value of $\sigma(j)$ which itself pointed to another index k, so swap index i and k. Continue this until $\sigma(i) = i$.

The solution to this problem consists of utilizing this fact of cycle sort. We know the array has size n, so there are values in the range $[1, n]$ and values outside this range. Put each value in the range to the correct index, and if there are any missing values in the range $[1, n]$ from the array, it will be filled up by those values outside the range. Therefore, after "sorting", the first instance where a value is not in its correct location, i.e. the index does not equal the content in the index, is the first missing positive value. If it happens that all are in their correct positions, then the first missing positive value is $n + 1$.

Container With Most Water

Given an integer array representing heights of length n, given two indices, the amount of water that can be held between the two is the largest rectangle that can "fit" the bounds of both height bars and spans the distance. More specifically, it is the area between two indices i and j, $i < j$, which is $= \min(h[i], h[j]) \cdot (j - i)$.

The maximum area can be found using a greedy approach. We prove this by contradiction. Suppose, to the contrary, that for some height array, the maximum area, indicated by two indices L and R, $L < R$, cannot be reached by this approach. Then the algorithm cannot reach the two potential states:

- $(L, R + 1)$, implying $h[L] > h[R + 1]$.
- $(L - 1, R)$, implying $h[R] > h[L - 1]$.

Regardless of the previous state, these states in turn are unreachable, since if they were reachable, then (L, R) is also reachable. These states are in turn have previous states feeding into it, and each previous state increases the distance between the two indices. If the length of the array is n, then going back $n-L+R$ states, all these states depend are generated from $(0, n - 1$, which is reachable, contradicting the assumption the other states are unreachable. Thus (L, R) is reachable.