# FINAL 450 SOLUTIONS

Descriptive Answers for all the problems.

# ARRAYS

# 1. Given an array, reverse the elements of the array.

Approach 1:

Make a new array. Traverse the original array from backside and copy the elements one by one into the new array. The new array is the reverse of the original array.

TC: O(n), SC: O(n)

Approach 2:

We can reverse the array in place itself. Make two pointers, one pointing to the first element and one pointing to the last element. Swap the two elements. Increment the first pointer, and decrement the second pointer. Do this until the left pointer is less than or equal to the right pointer. The array is now reversed.

TC: O(n), SC: O(1)

## Approach 3:

We can achieve the same using recursion. The recursive function would be:

reverse_array(array, first, last):

It swaps the elements at the first and the last pointers. Then the function is called for the remaining array, with first incremented and last decremented. The terminating condition would be first >= last. In the end, the array would be reversed.

# 2. Find the maximum and minimum elements in an array.

Approach 1:

Make two variables - min_ele = INT_MAX and max_ele = INT_MIN. Linearly go through the array and update the variables aptly. In the end, the correct result would be stored in the two variables.

TC: O(n)

Approach 2:

Tournament Method. This is a recursive approach. Find the max and min for the left half, and the same for the right half. Then true_max = max(left_max, right_max) and true_min = min(left_min, right_min).

TC: T(n) = T(floor(n/2)) + T(ceil(n/2)) + 2

On solving, we will get T(n) = O(n)

## Approach 3:

The approach is to compare elements in pairs. If the number of elements is odd initialize min and max as the first element, if even, then max as max of first two elements and min as min of first two elements. Then for every next two elements, find their min and max and compare with true_max and true_min and update the two accordingly. In the end, return true_max and true_min.

TC: O(n)

# 3. Kth Min and Max element of an array.

## Approach 1:

A naive approach is to sort the array and then return the kth element. This will be the kth smallest or largest element of the array.

TC: O(nlogn)

## Approach 2:

The best approach will be to use Quickselect algorithm. Quickselect is a modification of the quicksort algorithm where we place the pivot element at its correct place and stop the process when it is the kth element. Also, the process is only recurred for the left or right according to the value of the position of pivot.

TC: O(n^2) in the worst case and O(n) on average

# 4. Sort an array of 0s, 1s and 2s

Approach 1:

Naive approach is to count the number of zeros, ones and twos and then create a new array and place the numbers according to their counts.

TC: O(n), SC: O(n)

Approach 2:

Dutch National Flag Algorithm.

TC: O(n), SC: O(1)

# 5. Move all negative elements to one side of the array

Approach 1:

Apply the partition process of quicksort with 0 as the pivot element always. At the end, all the negative numbers will be on the left side and positive numbers on the right side.

TC: O(n), SC: O(1)

# 6. Find union and intersection of two sorted arrays.

Approach 1:

Merge algorithm can be used to find the union of the two sorted arrays, along with handling the duplicates in each array.

To find the intersection, use modified merge algorithm, where we increment the pointer if the element in one of the array is less than the other and when both are equal we print one of them.

TC: O(m+n) for both cases

# 7. Rotate an array cyclically by one.

Approach 1:

The simple approach is to store the last element. Then store the current element as prev_element and replace the next element with it and also modify the previous element. In the end, replace the first element with the last element that was stored earlier.

TC: O(n), SC: O(1)

# 8. Find the largest sum contiguous subarray

Approach 1:

Use Kadane’s Algorithm.

TC: O(n), SC: O(1)

# 9. Minimise the maximum difference between the heights

Variation 1: (The elements can be negative after adding or subtracting K)

Sort the array. The important observation is that, for some tower, if we choose to increase its height by K, then for all the towers to the left, we can increase their heights by K as well. And if we choose to decrease the height of a tower by K, then for all the towers to the right, their heights can be decreased by K as well.

Hence the problem reduces to find the minimum range for all the n points that divide the original array into two segments.

TC: O(n), SC: O(1)

Variation 2: (The elements should be non negative after performing the operations)

Traverse through the array and check if both subtract and add operations can make any difference to the minimum difference encountered so far.

TC: O(n), SC: O(1)

# 10. Minimum number of jumps to reach the end of an array.

Approach 1:

Dynamic programming approach. Make a jumps array that stores the minimum number of jumps required to reach a[i]. We can then return the last element of jumps array. Make two nested loops, in the inner loop, update the jumps[i] to the minimum of jumps[j] + 1 if jumps[j] is not -1 and arr[i] is reachable from j.

TC: O(n^2), SC: O(n)

Approach 2:

Ladder take and drop solution. We have a currentLadder that stores the index till which we can still go. nextLadder is the ladder that we keep hold of to use when our currentLadder is completely used. We then replace the currentLadder with the nextLadder and make nextLadder -1 again.

TC: O(n), SC: O(1)

# 11. Find duplicate in array of N+1 integers.

Approach 1:

On seeing abs(arr[i]), make element at i negative. If the element is already negative, that means the number has been visited. Return this as the repeated number.

TC: O(n), SC: O(1)

# 12. Merge two sorted arrays without using any extra space

Approach 1:

Use Gap Algorithm.

TC: O((m+n) * log(m+n)), SC: O(1)

# 13. Kadane’s Algorithm

Kadane’s Algorithm

TC: O(n), SC: O(1)

# 14. Merge Intervals

## Approach 1:

Sort the array of vectors. Make a new resultant vector. Make a counter for the array. Check if the current interval intersects with the last interval, then merge the interval. Return the final array of intervals as the result.

TC: O(n), SC: O(n)

# 15. Next Permutation

Approach 1:

Find the element just before the peak element of the array from the end. Find the first element from the back that is greater than that element. Swap both of them. Reverse the array from the peak element till the end.

TC: O(n), SC: O(1)

# 16. Count Inversions

## Approach 1:

The approach we follow is a modification of the merge algorithm. So divide the array into two equal parts and find the number of inversions for both the halves, let it be i1 for the left half and i2 for the right half. Then total number of inversions will be i1 + i2 + inversions found while merging the two arrays.

TC: O(n), SC: O(n)

# 17. Best time to buy and sell stock - one transaction

Approach 1:

Create a suffix array that stores the greatest element towards the right of every element of the array. Then traverse the stock prices and find the maximum profit that can be made.

TC: O(n), SC: O(n)

Approach 2:

Make a variable called minPrice. If a price lower than minPrice is found, update minPrice else find the difference between current price and min price and update the max profit that can be obtained.

TC: O(n), SC: O(1)

# 18. Count pairs with a given sum

## Approach 1:

Brute force approach will be to use two for loops and for each pair, check if its sum is equal to given sum. Then return the count of such pairs found.

TC: O(n^2), SC: O(1)

## Approach 2:

Make use of a hash map, where we store the count of each unique element in the array, then for each element, search for sum - element and add to count its frequency. Since each pair would be taken into account twice, return count divided by 2.

TC: O(n), SC: O(n)

# 19. Find common elements in 3 sorted arrays

Approach 1:

Apply intersection of two sorted arrays on the first and the second array. Then apply the same on the result and the third array, we will then have common elements from all the 3 sorted arrays.

TC: O(n1 + n2 + n3), SC: O(min(n1, n2, n3))

# 20. Rearrange array into alternating positive and negative integers

Variation 1: (Order of occurrence not important)

Partition the elements around 0. Then swap negative and positive elements alternately.

TC: O(n), SC: O(1)

Variation 2: (Order of occurrence is important)

Partition algorithm does not retain the order of the elements. The algorithm to follow, is find the first out of place element (negative at odd or positive at even), then find the first element with the opposite sign. Let the indices be i and j of both these elements. Right rotate the subarray from i to j (inclusive). In the end, the final array will be the required array.

TC: O(n^2), SC: O(1)

# 21. Find if there is any subarray with sum equals zero

Approach 1:

If the sum of a subarray from i to j is zero, then we can say that prefix_sum[i] = prefix_sum[j]. Hence, we calculate and store the prefix sums for each element of the array and if a preifx_sum has been seen earlier that means, there exists a subarray with sum equals zero.

TC: O(n), SC: O(n)

# 22. Find factorial of a large number

Approach 1:

Finding the factorial of very large numbers is not easy as it can have huge number of digits. The way to solve this problem is to use an array for multiplication. We iteratively multiply the numbers till n, and keep updating the result of multiplication in an array. The array is then printed in a reverse manner to return the correct result.

TC: O(n), SC: O(max_number_of_digits_in_factorial)

# 23. Find maximum product subarray

Approach 1:

This needs a modified Kadane's algorithm as product with negative numbers can make a previous minimum product maximum by making it positive. So, the way out is to have two variables - maxCurrentProduct, minCurrentProduct. Update the two as we do in Kadane's algorithm, but if a negative element is seen, then before performing the operations, swap the two. In the end return the maximum product encountered so far.

TC: O(n), SC: O(1)

# 24. Find the longest consecutive subsequence

Approach 1:

Sort the array. Remove all the duplicate elements. Make two variables - currentLength and maxLength. Increment currentLength by 1 on seeing consecutive elements and update maxLength when the sequence breaks. Return maxLength.

TC: O(nlogn), SC: O(n)

Approach 2:

Check for each element if it can be a start element or not. Use a hashmap to store all the unique elements present in the array. Then for each element of the array, check if its element - 1 is present or not. If not present, then this would be a start element. Then continue increasing currentLength till the consecutive sequence continues. Return the maximum of all currentLength found till the end.

TC: O(n), SC: O(n)

# 25. Find all elements that appear more than n/k times in an array of size n and a given number k

Approach 1:

Sort the array. Then easily count the frequency of each element and check if its count > n/k. Return the found elements.

TC: O(nlogn), SC: O(1)

Approach 2:

Tetris Method. Create a temp array of size k-1. Traverse through the original array and keep filling the temp array with struct of elements. If a new element is encountered, then remove the bottom row of the tetris, i.e. decrease the counts of all elements by 1 and skip the current element. At the end, the temporary array has potential elements. Check for each potential element, if it has a count > n/k.

TC: O(nk), SC: O(k)

# 26. Maximum profit by buying and selling a share at most twice

Approach 1:

Maximum profit = max{max_profit_one(0, i) + max_profit_one(i+1, n)}

Make two nested loops and check for each possible division the profit. Return the maximum profit found.

TC: O(n^2), SC: O(n)

Approach 2:

Improve upon the first approach using dynamic programming. Store the intermediate results. Store the results for max profit using one transaction in the left subarray from 0 to i and for the right subarray from i+1 to n. Then find the max possible sum using the above formula and return the same.

TC: O(n), SC: O(n)

# 27. Find if an array is a subarray of another array

## Approach 1:

Sort both the arrays. For every element of array 2, binary search array 1. If all elements are present, return true.

TC: O(mlogm + nlogn), SC: O(1)

## Approach 2:

Make hash map for the first array, storing the frequency of each element present. Then traverse through the second array decreasing the frequency of the same in the hash table. If all elements are found return true else return false.

TC: O(max(n, m)), SC: O(n)

# 28. Find triplets that sum to a given value in array

Approach 1:

Sort the array. For every element of the array, in the remaining array search for the sum - element using two pointer technique for a sorted array.

TC: O(N^2), SC: O(1)

Approach 2:

Hashing based solution. For every element apply the hash-based method to search for sum - element pairs in the remaining array.

TC:O(N^2), SC: O(N)

# 29. Trapping Rain Water Problem

Approach 1:

The approach is to find how much water will be there over every block. Make two auxiliary arrays and in the first array store the greatest height block to the right of each array element. In the second array, do the same towards the left side.

Then for each element, the amount of water over it = min(left_max[i], right_max[i]) - heigh[i].

Return the sum of water holdings.

TC: O(N), SC: O(N)

# 30. Chocolate distribution problem

## Approach 1:

The main observation is that, to minimise the difference between extreme values, we need to consider consecutive elements in a sorted array. So we use the sliding window algorithm. After sorting the array, we slide the window of size m (number of students). Then we store the minimum difference found so far.

TC: O(nlogn), SC: O(1)

# 31. Smallest subarray with sum greater than a given value

Approach 1:

Nested loop through the array and check for every subarray its sum using prefix and suffix arrays. Then return the size of the smallest subarray having sum greater than the given value.

TC: O(N^2), SC: O(N)

# 32. Three way partitioning of the array

Approach 1:

Use the dutch national flag algorithm.

TC: O(N), SC: O(1)

# 33. Minimum swaps required to bring elements less than K together in an array

Approach 1:

The approach is to use sliding window. First count the number of good elements, that is elements less than equal to K. Then make a window of size good elements and count the number of bad elements in that window. Then start sliding the window, updating the number of bad elements. Return the minimum bad element count found so far. That will be the minimum number of swaps required.

TC: O(N), SC: O(1)

# 34. Minimum merges to make an array palindrome

Approach 1:

Let f(i, j) denote the minimum merges from i till j. If arr[i] == arr[j], then result will be f(i+1, j-1). If arr[i] > arr[j], apply merge operation at j and return 1 + f(i, j-1) else return 1 + f(i+1, j).

TC: O(N), SC: O(1)

# 35. Median of two sorted arrays of equal size

Approach 1:

The approach is to compare the middle elements. If they are equal, then return one of them as the median. If first one is greater than the second one, then the median must be from first half of first array and second half of second array. If second is greater than the first one, then median must be from second half of first array and first half of second array.

TC: O(logn), SC: O(1)

# 36. Median of two sorted arrays of different sizes

Approach 1:

The approach is to keep the first array smaller. If size of first array is zero, return the median of the second array. Keep two pointers on the first array. One at 0 index and other at n index, as there are n+1 partitions possible for an array of size n. Then find the middle partition using (low + high) / 2. Calculate the partition of the second array, such that number of elements to the left are equal to those on the right. Compare the middle four elements to determine the median and appropriate changes to low and high.

TC: O(min(logm, logn)), SC: O(1)

# MATRIX

# 1. Spiral traversal of a matrix

Approach 1:

Recursively call the function to print the first row, the last column, the last row and then the first column and then update the start point and the rows and columns appropriately.

TC: O(N*M), SC: O(N*M)

# 2. Search in a sorted matrix

Approach 1:

Treat the 2D matrix as a single sorted list and apply binary search on it. The only modification is that, we need to convert the single list index to the matrix index to fetch the corresponding element.

TC: O(log(N*M)), SC: O(1)

# 3. Find median in a row sorted matrix

Approach 1:

The approach is to find the minimum and the maximum elements of the matrix. Then binary search over the search space and check for the mid element, if the number of elements less than it is equal to desired number then increase min to mid + 1 else decrease max to mid. Finally return the min element as it will converge to the result.

TC: O(nlogn), SC: O(1)

# 4. Find row with maximum number of 1s

Approach 1:

Search for the first one in every row. Then update the max, if total columns minus index of first one is greater than the previous value. Return the maximum count.

TC: O(n logm), SC: O(1)

Approach 2:

Find first one in the first row. For every other row, update the left index if a one is found towards its left. Continue, till the maximum count of ones is found.

TC: O(m + n), SC: O(1)

# 5. Print elements in sorted order for a row and column wise sorted matrix

Approach 1:

Make a min heap of size number of rows of the matrix. Print and pop the minimum element from the heap and put the next element from the same row to the heap.

This way, we will have a sorted list printed.

TC: O(N^2logN), SC: O(N)

# 6. Maximum size rectangle

Approach 1:

Use the maximum area under histogram method for each row of the matrix, then return the maximum area found until end as the answer. For max area under histogram, create a stack and keeping pushing heights till it is greater than the height at the top of the stack, else keep popping the elements and updating the maximum area found so far.

TC: O(N^2), SC: O(N)

# 7. Find specific pair in a matrix

Approach 1:

Brute force method is to consider all possible (a, b) and (c, d) pairs and return the maximum difference found between the elements.

TC: O(N^4), SC: O(1)

Approach 2:

The approach is to use extra space and store the maximum element from i, j to n-1, n-1 in an auxiliary matrix. Then for each element of the original matrix find the difference and update the maximum difference found so far. Return this as the answer.

TC: O(N^2), SC: O(N^2)

# 8. Rotate matrix by 90 degrees

## Approach 1:

Take transpose of the matrix and then take the mirror image, this will give the rotated matrix by 90 degrees.

TC: O(N^2), SC: O(1)

## Approach 2:

Rotate the matrix around the main diagonal. Then rotate one more time along the middle column of the matrix, which will give the resultant matrix.

TC: O(N^2), SC: O(1)

# 9. Kth element in a row and column wise sorted matrix

Approach 1:

Use a min heap to store the first column on the matrix. Then keep on popping and adding elements till the kth element is found. Print the element.

TC: O(k logn), SC: O(n)

Approach 2:

Use the binary search over space method, start with min and max pointers, keep narrowing min pointer to the result.

TC: O(nlogn), SC: O(1)

# 10. Common elements in all rows of a given matrix

## Approach 1:

Sort all the rows. Then taking pairs of rows, find the common elements using the modified merge procedure. In the end, return the common elements found so far.

TC: O(nlogn), SC: O(n)

## Approach 2:

Make a hashmap for the first row. Then traverse all the rows to find common elements. Return the list of common elements found.

TC: O(n), SC: O(n)

# STRING

# 1. Reverse a string.

Approach 1:

Same approach as reversing an array, either iteratively or recursively.

TC: O(n), SC: O(1)

# 2. Check if a string is a palindrome

Approach 1:

Iterative method, compare the mirror elements till the middle element. If all are the same, then return true.

TC: O(n), SC: O(1)

Approach 2:

Recursive method, compare the first and the last elements, return isPalindrome(i+1, j-1).

TC: O(n), SC: O(1)

# 3. Find duplicate characters in a string.

Approach 1:

Use a hashmap to keep track of characters that have been seen. Then for every duplicate character seen print it.

TC: O(N), SC: O(N)

# 4. Why strings are immutable in java

The string is Immutable in Java because String objects are cached in the String pool. Since cached String literals are shared between multiple clients there is always a risk, where one client's action would affect all another client.

# 5. Check if a string is a rotation of another string

Approach 1:

Concatenate str1 to itself. If str2 is a substring of the concatenated string, then return true else return false. To check if str2 is a substring, we can use KMP algorithm or inbuilt C++ find function.

TC: O(m), SC: O(m)

# 6. Check if a string is a valid shuffle of two strings

Approach 1:

Traverse through the result string and if the character matches with the first string increase both pointers. Do similarly for both the strings, if at any time, the conditions do not meet, return false. In the end return true.

TC: O(n+m), SC: O(1)

# 7. Count and say

Approach 1:

Make a function that will tell how to say a string. SayString(string s). Then in the countAndSay function, return SayString(countAndSay(n-1)), if n > 1 else return “1”.

TC: O(n), SC: O(n)

# 8. Longest Palindromic Substring

Approach 1:

Use dynamic programming. Calculate if a substring is a palindrome from i to j. Loop over the length of the substring from 3 to n. Also keep updating the maximum length of the substring that is a palindrome.

TC: O(N^2), SC: O(N^2)

# 9. Longest repeating subsequence

Approach 1:

The approach is to make use of the LCS algorithm. For the given string str, find LCS(str, str), with the modification that the indices of the matching characters should not be the same. Return the length of the maximum LCS found so far.

TC: O(N^2), SC : O(N^2)

# 10. Find all subsequences of a string

Approach 1:

Simple approach is to either consider an element in the output or not consider it, so make recursive calls with each case.

TC: O(2^n), SC: O(1)

# 11. Print all permutations of the given string

Approach 1:

For every element in the string, swap the first element with that character and then call get_permuations from the next index. When index points to the last character, print the string, which will be the terminating condition of the recursion.

TC: O(n!), SC: O(1)

# 12. Split binary strings into maximum substrings with equal number of 0s and 1s

Approach 1:

A very simple approach is to keep two variables, counting ones and zeros. Whenever the count of zeros and ones become equal, increment the number of substrings by one. Return the final count.

TC: O(n), SC: O(1)

# 13. Word wrap problem

## Approach 1:

We follow the dynamic programming approach. The first step is to calculate the extra spaces 2D matrix, storing the extra spaces for words from i to j stored in a single line. Then from these extra spaces, we calculate the cost array from i to j similarly. Then we use the recursive formula, c[j] = for i from 1 to j, min(c[i-1] + lc[i][j]), where both of them are not equal to infinity. Along the way keep storing the break indices as well, to print the final output.

TC: O(N^2), SC: O(N^2)

# 14. Edit distance

Approach 1:

Dynamic programming approach, where we start to find the edit distance for subsets of the two strings. The value of dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1]dp[j-1] + 0/1 ( depends if str1[i-1] == str2[j-1] or not )).

Return the value of dp[n][m].

TC:O (N^2), SC: O(N^2)

# 15. Find next number with same set of digits

Approach 1:

The same logic as the next permutation problem.

1) Find the first smaller number from the end. Call this as the peak element.

2) From the end, find the first number greater than this peak element.

3) Swap the two elements, then reverse the list from index onwards, excluding.

TC: O(N), SC: O(1)

# 16. Balanced parenthesis problem

Approach 1:

Make a stack. If the bracket is a opening one, push it into the stack, else check if top of stack has a matching opening brace, pop it from the top and move forward with the input. If in the end, the stack is empty, then return true as the string is a valid one.

TC: O(N), SC: O(N)

# 17. Word break problem

## Approach 1:

The approach is to iteratively increase the index and for every i, return true if substr(0, i) is found in dictionary and wordBreak(i+1, j) returns true.

TC: O(N^2), SC: O(1)

# 18. Rabin Karp Algorithm

Approach 1:

Rabin Karp Algorithm

# 19. KMP Algorithm

Approach 1:

KMP Algorithm

# 20. Convert a sentence into its equivalent mobile numeric keypad sequence

Approach 1:

For each character, store what string sequence of numbers are there. Then traverse through the whole string and print the sequence stored for each character.

TC: O(N), SC: O(1)

# 21. Minimum bracket reversals to make string balanced

Approach 1:

1) Remove the valid string from the input, the way to do this is to use a stack and keep popping matching pairs of brackets.
2) The stack has now string of type }}}}}{{{{{{{{{{..., count the number of open and close brackets.
3) The minimum reversals needed will be ceil(open/2.0) + ceil(close/2.0).

   TC: O(N), SC: O(N)

# 22. Count palindromic subsequence

Approach 1:

The recursive approach will be:

1) If length of string is 1, then return 1.
2) If str[i] == str[j], then return count(i+1, j) + count(i, j-1) + 1.
3) else, return count(i+1, j) + count(i, j-1) - count(i+1, j-1).

After optimising the recursion using dynamic programming, the

TC: O(N^2), SC: O(N^2)

# 23. Count number of given string, in a 2D character array

Approach 1:

Follow a recursive approach, where for each character in the matrix, if it matches with the current character, then check in the all four directions for the next character of the pattern. Increase count whenever the pattern becomes empty. Return the final count as the answer.

TC: O(N^4), SC: O(1)

# 24.Boyer Moore algorithm for pattern searching

Approach 1:

Boyer Moore Algorithm

# 25. Converting roman numerals to decimal

Approach 1:

The key observation is that, if a greater symbol follows a smaller symbol, then it can be simply added to the result, but if a smaller symbol occurs before a greater symbol then it needs to be subtracted from the result. Like if first symbol is s1 and second is s2, then res += s2 - s1.

TC: O(N), SC: O(1)

# 26. Longest common prefix

Approach 1:

Take the longest common prefix as the first string, then keep on finding the longest common prefix with the rest of the strings. Return the last LCP.

TC: O(S), where S is the sum of characters in all the strings, SC: O(1)

# 27. Number of flips to make binary string alternate

Approach 1:

The important observation is that either the final alternate string can start with a 0 or 1. So we compare the input string with both the strings and return the minimum number of differences found in whichever case.

TC: O(N), where N is the length of the string, SC: O(1)

# 28. Find the first repeated word in string

Approach 1:

Make a hash map and store the frequency of each word in the string, return the second most occurring word using that hashmap count.

TC: O(N), SC: O(N)

# 29. Minimum swaps for bracket balancing

Approach 1:

The approach is to make a count and a number of swaps variable. Increase the count on seeing a [ brace and decrease it on ] brace. Whenever the count becomes negative, swap the current closing brace with the next opening brace. To make the solution optimal, we can store the indices of all the opening braces. Increase the count by j-i, where j is the index if next opening brace.

TC: O(N), SC :O(N)

# 30. Find LCS of two strings

Approach 1:

LCS Algorithm

# 31. Program to generate all possible valid IP addresses from a given string

Approach 1:

Make three for loops showcasing the place where the dot can be placed in the string. Then for each part of the ip made, validate it and proceed, if a valid ip is found, then add it to the vector. Return the final vector.

TC: O(N^3), SC: O(N)

# 32. Find the smallest window that contains all the characters of the string itself

Approach 1:

Count all the distinct characters in the input string. Start with a sliding window of size 1. Then keep increasing the size till it contains all the distinct characters. After that start removing extra characters in the beginning. Update the minimum size of the window found so far.

TC: O(N), SC: O(N)

# 33. Rearrange characters, so that no two adjacent characters are the same in a string

Approach 1:

The main thing to note here is that, we start putting the character with the highest frequency first and then take the next highest frequency and continue. We can use a max heap for this process.

TC: O(NlogN), SC: O(N)

Approach 2:

We can fill firstly the even positions of the output string starting with the highest frequency character, then start filling the remaining even positions and then the odd positions of the string.

TC: O(N), SC: O(N + 26)

# 34. How many characters need to be added in front to make a string palindrome

Approach 1:

The number of characters that need to be added will be equal to the original string length - length of the string that is already a palindrome. Start removing characters from the end and check if the remaining string is a palindrome, repeat the process and return the number of characters removed.

TC: O(N^2), SC: O(1)

Approach 2:

This question can be solved using LPS array as well. Concatenate the reverse of the string to the original string separated by a $ sign. e.g. AAB$BAA. Find the LPS of the whole original string, then return the answer as original string length - LPS last element.

TC: O(N), SC: O(N)

# 35. Group all anagrams together

## Approach 1:

The intuition is that all anagrams when sorted give the same string. Make a map with key as sorted version of the string, push the anagrams list as value.

TC: O(NK log K), SC: O(NK)

## Approach 2:

Instead of taking the sorted string as a key, we can make a string containing the count of all characters separated by #. The other steps remain the same.

TC: O(NK), SC: O(NK)

# 36. Find the smallest window that contains all characters of another string

Approach 1:

Same approach as question 32 Strings, with a modification that we need to keep track of all the characters that occur.

TC: O(N), SC : O(N)

# 37. Remove consecutive characters

Approach 1:

Simple iterative approach, add a character only once to the resultant string. Return the resultant string.

TC: O(N), SC: O(1)

# 38. Wildcard string matching

## Approach 1:

This problem has a nice recursive solution that can be optimised using dynamic programming. If both the characters match or character in pattern is a ?, then return for the remaining parts of the two strings. If a * comes, then return for i-1, j or i, j-1.

TC: O(N^2), SC: O(N^2)

# 39. Find number of customers who could not get a computer

Approach 1:

Maintain a seen array, where we keep track of which user has been seen and which user is currently using a computer. Accordingly calculate the number of users, who could not get a computer. It is provided that, a person who does not get a computer always leaves before the person who got one.

TC: O(N), SC : O(256)

# 40. Transform one string to other using minimum number of given operation

Approach 1:

The logic is to start matching characters from the back, if the characters match reduce the problem to n-1 size. Else check the position of the mismatched character. This seems like a N^2 operation but with a clever approach can be N in the worst case. Make two pointers i and j, if characters match decrement both, if they don't increment result and decrement only i. Do this until i >= 0. Return result.

TC: O(N), SC: O(1)

# 41. Check if two strings are isomorphic

Approach 1:

Make two maps and store each corresponding character as key and value both. Check for the whole length of the string if each coming character maps to the correct character it mapped previously to.

Tc: O(N), SC: O(N)

# 42. Print all sentences that can be formed from a list of words

Approach 1:

Simple recursion problem, where we iterate through the current row and for each word append it to the output. The parameters of recursion would be the list of words, the current row, the output string so far. Whenever the current row equals total number of rows, print or add the output string.

TC: O(L1 * L2 * ... Lk), where k is the number of rows, SC: O(1)

# SEARCHING AND SORTING

# 1. Find first and last occurrences of x in a sorted array

Approach 1:

Modified binary search algorithm, where for the first occurrence we compare it with the left element and if not equal return it as the answer, similarly for the last occurrence, compare with the right element.

TC: O(N), SC: O(1)

# 2. Find fixed point in a given array

Approach 1:

Since the array is not sorted, we have to search linearly for such an element, whose value is the same as its index.

TC: O(N), SC: O(1)

# 3. Search in a rotated sorted array

Approach 1:

First find the pivot in rotated sorted array. Compare the target element with the last element, if larger then search from 0 to pivot-1 else search from pivot to end. Return the index.

TC: O(logN), SC: O(1)

# 4. Square root of an integer

Approach 1:

The problem reduces to finding the square root of a number. This can be improved to logN time using the square root finding algorithm.

TC: O(logN), SC: O(1)

# 5. Maximum and minimum of an array using minimum number of comparisons

Approach 1:

Can be solved using tournament method or compare in pairs method. In the tournament method, find the min and max for both halves of the array, then compare the pairs and return the final answer. The base case will be when array size is 1 or two.

TC: O(N), SC: O(1)

# 6. Optimum location of point to minimise total distance

Approach 1:

We use the ternary search here since the distance curve will be a unimodal function.

TC: O(logN), SC: O(1)

# 7. Find repeating and missing

Approach 1:

Use the negative value at visited index method to find the repeating element, then the index at which the element is still positive will be the missing element. Return the array of the two elements as the result.

TC: O(N), SC: O(1)

# 8. Find majority element

Approach 1:

Tetris method, where the temporary array will now have a size of K-1 i.e. 1, because K = 2 in this case.

TC: O(N), SC : O(1)

# 9. Searching in an array where adjacent differ by utmost K

Approach 1:

Start searching for the element using linear search. The important modification here is that if the element is having a diff difference from x, then we can jump over to diff/k elements to the right.

TC: O(N), SC: O(1)

# 10. Find a pair with a given difference

Approach 1:

A naive approach is to use two nested loops with N^2 time complexity. A better approach is to sort the array, then for each element in the array, binary search for the other remaining element, such that the difference comes out as desired. This can be further improved by using two pointer approach, where we keep two pointers next to each other and keep checking for their difference and updating the pointers.

TC: O(NlogN), SC: O(1)

# 11. Find four elements that sum to a given value

Approach 1:

This problem can be solved in quite a few ways. One of the approaches is to make two pointers, find the pair sum and then in the remaining array find the pair with remaining sum using two pointer approach. While incrementing the pointers, we take care of duplicates (skip them).

TC: O(N^3), SC: O(1)

# 12. Maximize sum such that no two elements are adjacent

## Approach 1:

Use recursive approach and convert to top down dynamic programming method. The recursive approach will be either to steal from a house or not and compute the profit that can be made in each case. Store the intermediate results for a better time complexity.

TC: O(N), SC: O(N)

## Approach 2:

Make two variables, val1 and val2, that store the max loot for one house and first two houses respectively. Then for each remaining house, store the max of val1 + currElement and val2. Replace val1 with val2 and val2 with max. Return the max loot in the end.

TC: O(N), SC: O(1)

# 13. Count triplet with sum smaller than a given value

Approach 1:

Sort the array. For every element, find sum - arr[i]. We need to find number of pairs in the remaining array having sum less than this remainder sum. We can use the two pointer approach for this. If sum is less than given remainder sum, then increase count by end - start, else decrease the end pointer.

TC: O(N^2), SC: O(1)

# 14. Merge two sorted arrays

Approach 1:

Gap Algorithm

TC: O((n+m)log(n+m)), SC: O(1)

# 15. Print all subarrays with zero sum

Approach 1:

Make a prefix array. Make a hashmap to store seen sums. If the prefix sum is zero then increase count by 1. Also, if the prefix sum has been seen earlier, then increase the count by the frequency stored in the hashmap for that sum. Return the total count.

TC: O(N), SC: O(N)

# 16. Product array puzzle

Approach 1:

Make a prefix product array of size n. The ith index stores the product till the i-1th indices. Then for each element of the array, find the suffix product that contains product from ith to the n-1th index. The result array would have product of prefix element and suffix element at ith index. Return the final array.

TC: O(N), SC: O(N)

# 17. Sort array according to set bit count

Approach 1:

For each element in the array, find the number of set bits using the count set bits algorithm. Make a hashmap of size 32. Save the elements corresponding to the bit count in the hashmap. Then for each key of the hashmap, start printing the elements one by one from the start, the elements will be in the sorted order automatically. Finding the number of set bits in each number takes logN time.

TC: O(N logN), SC: O(N)

# 18. Minimum swaps required to sort the array

Approach 1:

A very interesting approach is to see every node in the array as a node in the graph. Then make directed edges from every element, to other nodes where it should have been, if it was a sorted array. We will have some cycles in the graph. Make an array that contains pair of element and its position. Sort the array. If an element has been visited or is in the correct position, skip it. Else, try to make a cycle, by marking each element as visited and moving on till an already visited element is found.

TC: O(NlogN), SC: O(N)

# 19. Bishu and Soldiers

## Approach 1:

Simple approach is to add every element that is less than equal to x and add it to the result. Print the index of the element and the sum of elements less than equal to x.

TC: O(QN), SC: O(1)

# 20. Kth smallest number again

Approach 1:

The approach is to first make all the ranges non overlapping and sorted. We can easily do this using the merge intervals strategy. After doing this, for every range, check if K <= B - A + 1, i.e. if Kth element falls into that range, if it does return A + K - 1. If it doesn't update K to K - (B - A + 1), i.e. remove the size of the previous range from K.

TC: O(N log N + Q * N), SC: O(N)

# 21. Find pivot in a sorted and rotated array

Approach 1:

Check if arr[0] < arr[n], if yes return 0. Make two pointers, one start and one end. For the mid element, check if it is less than the first element. If not, then update start to mid + 1. Else, check if previous element is greater than current element. If yes, return mid else update end to mid - 1.

TC: O(log N), SC: O(1)

# 22. Kth element of two sorted arrays

Approach 1:

A naive approach would be to follow the merge operation. Track the counter and whenever counter becomes equal to K, return the just added previous element.

TC: O(K), SC: O(1)

Approach 2:

We can perform better using a divide and conquer approach. Keep two pointers on both the sorted arrays. Find the mid elements of both the arrays. Find the total length of arrays till the mid element. Compare it with K, in both cases compare the middle elements and update the two pointers accordingly. This will be a recursive approach.

TC: O(log N + log M), SC: O(1)

# 23. Aggressive Cows

## Approach 1:

A naive approach would be to check for all possible configurations, and return the best answer, but it would be a factorial approach. A better way is to search over the space. The search space would be the minimum separation possible between cows, varying from 0 to N-1, where N is the number of stalls. We binary search over the sorted stalls array and for each element, check if that separation is possible.

TC: O(NlogN), SC: O(1)

# 24. Book Allocation Problem

Approach 1:

Binary search over space. The space would vary from 0 to sum of pages in all books. For each binary searched pages, check if it is possible to divide the books in the students in such a way so that the maximum possible pages comes out to be the same. Narrow down over the search space to find the answer.

TC: O(NlogN), SC: O(1)

# 25. EKOSPOJ

## Approach 1:

Binary search over the possible solution space, i.e from 0 to max height of the tree. For each element, check if it is possible to get desired wood by setting the saw blade at that height.

TC: O(N logN), SC: O(1)

# 26. Job Scheduling Algorithm

Approach 1:

This problem has a recursive solution, where we have two possibilities of either including or excluding a job. If we include a job, we need to find the first disjoint job from the end. This recursive approach can be optimised through top down dp approach. To find the first non conflicting job, we can use binary search.

TC: O(N logN), SC: O(N)

# 27. Missing number in AP

Approach 1:

Using the nth term formula of A.P., we can determine the condition, if a number exists in the sequence or not. Take care of corner cases, like 0 common difference etc.

TC: O(1), SC: O(1)

# 28. Smallest number with at least n trailing zeros in factorial

Approach 1:

For a number's factorial to have n trailing zeros, the max number would be 5*n. So, we can search from 0 to 5*n and for each element, test if the number of trailing zeros are less or greater than equal to the required number. Accordingly traverse the search space. Number of trailing zeros in x! = x/5 + x/25 + x/125 + ....

TC: O(logN), SC: O(1)

# 29. Painter’s partition problem

Approach 1:

Search over the solution space from 0 to sum of length of all the boards. For each possible time, check if it can be achieved using K painters through the sequential assigning approach.

TC: O(N logN), SC: O(1)

# 30. Roti Prata SPOJ

## Approach 1:

Search over the solution space from 1 to the time taken to cook P paratas by the highest ranked cook. For each time, check if it is feasible to cook P paratas in that time. Try to improve the solution by modifying the search pointers accordingly.

TC: O(N logN), SC: O(1)

# 31. Double Helix SPOJ

Approach 1:

Use a modified merge algorithm. Till you get the similar elements in the arrays, store the sums of the two arrays in two different variables. Add to the final sum, whichever is greater of the two. Keep doing this until the end of the arrays.

TC: O(max(n, m)), SC: O(1)

# 32. Subset Sums

## Approach 1:

A naive approach would be to check for all subsets their sums and if they lie in the desired range, increase the count. TC of this approach would O(2^n). We can reduce this by a factor of two. Divide the array into two halves. For each half, find the arrays storing sums of all possible subsequences. Sort the second array. Then linearly traverse through the first array and binary search in the second array appropriately, increasing the count.

TC: O(2^(n/2)), SC: O(N)

# 33. Find the inversion count

## Approach 1:

Same approach as discussed in the arrays section.

# 34. Implement merge sort in place

Approach 1:

Do the standard merge sort procedure. We only need to change the merge part. Instead of traditionally using an auxiliary array for the merge part, we will use the gap algorithm for merging the arrays in place.

TC: O(logN * N logN), SC: O(1)

# LINKED LIST

# 1. Reverse a linked list

## Approach 1:

Iterative approach can be implemented using three pointers, curr, prev, and next. Keep modifying until curr becomes null. Make head equal to prev.

## Approach 2:

Recursive approach, call reverse on head->next, then do appropriate operations to link the head to the reversed list and then return the head of the reversed list.

TC: O(n), SC: O(1)

# 2. Reverse a linked list in groups of given size

Approach 1:

Use the iterative three pointer approach to reverse the linked list along with a counter to count the number of nodes that have been reversed. Then recursively call the reverse function on the next pointer. And set the head->next to the result.

TC: O(N), SC: O(1)

# 3. Detect a loop in a linked list

Approach 1:

Use the floyd's cycle finding algorithm. Make two pointers slow and fast. Make slow = slow->next and fast = fast->next->next. If at some point, both slow and fast point to the same node, then return true else if the list ends, return false.

TC: O(N), SC: O(1)

# 4. Remove loop in a linked list

Approach 1:

Detect a cycle using the floyd's cycle finding algorithm. Make the slow pointer at head again and keep moving slow and fast at the same pace. They first meet at the starting point of the loop. Make the last node of the loop point to null. The loop is now removed from the linked list.

TC: O(N), SC: O(1)

# 5. Find the starting point of the loop

Approach 1:

Use the same approach as the last problem.

# 6. Remove duplicates from a sorted linked list

Approach 1:

Make two pointers, prev and temp. Keep moving temp till similar nodes are being found. Set the next of prev to temp whenever new node is found. Stop when prev equals NULL.

TC: O(N), SC: O(1)

# 7. Remove duplicates from an unsorted list

Approach 1:

Traverse through the linked list and keep adding the nodes data to the hashmap. Then only add unseen nodes to a new linked list.

TC: O(N), SC: O(1)

# 8. Move node from last to front of a linked list

Approach 1:

If length of list is >= 2, then make two pointers curr and prev that point to the last and second last elements of the linked list respectively. Make curr->next = head, prev->next = NULL and head = curr. Return the head.

TC: O(N), SC: O(1)

# 9. Add 1 to a number represented by a linked list

Approach 1:

Find a pointer to the rightmost non nine digit in the list. If no such node found, then return the list nodes as 0 and 1 appended at the front. If last node is the required node, simply add 1 and return. Else, add 1 to the current node and make all the other right nodes data to zero.

TC: O(N), SC : O(1)

# 10. Add two numbers represented by linked lists

Approach 1:

Reverse the linked lists if they are not already. Make two pointers to both the lists. Start adding the digits, keeping track of carry also. Then do the same for the remaining portions of any list. If carry is 1 at the end of the operations add a node of 1 in the front. Return the new linked list formed.

TC: O(n+m), SC: O(1)