

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/branchSums/solution1/branchSums.kt

```
package ae.easy.branchSums.solution1

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun branchSums(root: BinaryTree): List<Int> {
    val result = mutableListOf<Int>()
    dfsHelper(root, 0, result)
    return result
}

fun dfsHelper(node: BinaryTree, currentSum: Int, result: MutableList<Int>) {
    if (node.left == null && node.right == null) {
        result.add(currentSum + node.value)
        return
    }

    if (node.left != null) {
        dfsHelper(node.left!!, currentSum + node.value, result)
    }

    if (node.right != null) {
        dfsHelper(node.right!!, currentSum + node.value, result)
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/bubbleSort/solution1/bubbleSort.kt

```
package ae.easy.bubbleSort.solution1

fun bubbleSort(array: MutableList<Int>): List<Int> {
    // At the point our sorting loop didn't have any swap, we are done and
    // our array is sorted.
    // This variable will facilitate that
    var isSorted = false

    // This variable is used for optimization.
    // After each iteration the last element in the array is the largest
    // element (i.e. in it's sorted position)
    // So we don't need to consider it in the next iteration. It's already
    // in it's appropriate place. To facilitate this, we use this counter variable
    var counter = 0

    while (!isSorted) {
        isSorted = true

        // We need to loop till array.size() - 1 - counter
        // array.size() - 1 because we will compare i with i + 1
        // counter because in each step we have to skip the last element
        for (i in 0 until array.lastIndex - counter) {
            if (array[i] > array[i + 1]) {
                swap(i, i + 1, array)

                // In this step we made a swap. So the array is not sorted
                isSorted = false
            }
        }

        // After each step we increase the counter to count the number of
        // elements reached their sorted position so that we can skip them.
        counter += 1
    }

    return array
}

fun swap(i: Int, j: Int, array: MutableList<Int>) {
    val temp = array[i]
    array[i] = array[j]
    array[j] = temp
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/nodeDepths/solution1/nodeDepths.kt

```
package ae.easy.nodeDepths.solution1

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun nodeDepths(root: BinaryTree): Int {
    return helper(root, 0)
}

fun helper(node: BinaryTree?, depth: Int): Int {
    node ?: return 0

    return depth + helper(node.left, depth + 1) + helper(node.right, depth
+ 1)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/nodeDepths/solution2/nodeDepths.kt

```
package ae.easy.nodeDepths.solution2

import java.util.Stack

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun nodeDepths(root: BinaryTree): Int {
    val stk = Stack<Pair<BinaryTree?, Int>>()
    stk.push(root to 0)

    var sumOfDepths = 0

    while (stk.isNotEmpty()) {

        val (currentNode, currentDepth) = stk.pop()

        currentNode ?: continue

        sumOfDepths += currentDepth

        stk.push(currentNode.left to currentDepth + 1)
        stk.push(currentNode.right to currentDepth + 1)
    }

    return sumOfDepths
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/productSum/solution1/productSum.kt

```
package ae.easy.productSum.solution1

fun productSum(array: List<*>, multiplier: Int = 1): Int {
    var sum = 0;

    for (e1 in array) {
        if (e1 is List<*>) {
            sum += productSum(e1, multiplier + 1)
        } else {
            sum += e1 as Int
        }
    }

    return sum * multiplier
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/classPhotos/solution1/classPhotos.kt

```
package ae.easy.classPhotos.solution1

fun classPhotos(redShirtHeights: MutableList<Int>, blueShirtHeights: MutableList<Int>): Boolean {

    redShirtHeights.sort()
    blueShirtHeights.sort()

    if (redShirtHeights.size == 0) return true

    val firstRow = if (redShirtHeights[0] < blueShirtHeights[0])
        redShirtHeights else blueShirtHeights
    val secondRow = if (firstRow == redShirtHeights) blueShirtHeights else
        redShirtHeights

    for (i in 0 .. firstRow.lastIndex) {
        if (firstRow[i] >= secondRow[i]) {
            return false
        }
    }

    return true
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/binarySearch/solution1/binarySearch.kt

```
package ae.easy.binarySearch.solution1

fun binarySearch(array: List<Int>, target: Int): Int {
    return binarySearchHelper(array, target, 0, array.lastIndex)
}

fun binarySearchHelper(array: List<Int>, target: Int, left: Int, right: Int): Int {
    if (left > right) return -1
    val mid = (left + right) / 2
    val potentialMatch = array[mid]

    return if (potentialMatch == target) {
        mid
    } else if (target < potentialMatch) {
        binarySearchHelper(array, target, left, mid - 1)
    } else {
        binarySearchHelper(array, target, mid + 1, right)
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/binarySearch/solution2/binarySearch.kt

```
package ae.easy.binarySearch.solution2

fun binarySearch(array: List<Int>, target: Int): Int {
    var left = 0
    var right = array.lastIndex

    while (left <= right) {
        val mid = (left + right) / 2
        val potentialMatch = array[mid]

        if (target == potentialMatch) {
            return mid
        } else if (target < potentialMatch) {
            right = mid - 1
        } else {
            left = mid + 1
        }
    }

    return -1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/nthFibonacci/solution1/nthFibonacci.kt

```
package ae.easy.nthFibonacci.solution1

fun getNthFib(n: Int): Int {
    if (n == 1) {
        return 0
    }

    if (n == 2) {
        return 1
    }

    return getNthFib(n - 1) + getNthFib(n - 2)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/nthFibonacci/solution2/nthFibonacci.kt

```
package ae.easy.nthFibonacci.solution2

fun getNthFib(n: Int): Int {
    val memo = mutableMapOf(1 to 0, 2 to 1)
    return helper(n, memo)
}

fun helper(n: Int, memo: MutableMap<Int, Int>): Int {
    if (memo.containsKey(n)) {
        return memo[n]!!
    }

    memo[n] = helper(n - 1, memo) + helper(n - 2, memo)

    return memo[n]!!
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/nthFibonacci/solution3/nthFibonacci.kt

```
package ae.easy.nthFibonacci.solution3

fun getNthFib(n: Int): Int {
    val lastTwo = mutableListOf(0, 1)

    var counter = 3

    while (counter <= n) {
        val nextFibonacci = lastTwo[0] + lastTwo[1]
        lastTwo[0] = lastTwo[1]
        lastTwo[1] = nextFibonacci

        counter += 1
    }

    return if (n > 1) lastTwo[1] else lastTwo[0]
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/twoNumberSum/solution1/twoNumberSum.kt

```
package ae.easy.twoNumberSum.solution1

fun twoNumberSum(array: MutableList<Int>, targetSum: Int): List<Int> {
    val numSet = mutableSetOf<Int>()

    array.forEach { currentNum ->
        if (numSet.contains(targetSum - currentNum)) {
            return listOf(currentNum, targetSum - currentNum)
        }

        numSet.add(currentNum)
    }

    return listOf()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/twoNumberSum/solution2/twoNumberSum.kt

```
package ae.easy.twoNumberSum.solution2

fun twoNumberSum(array: MutableList<Int>, targetSum: Int): List<Int> {
    array.sort()

    var left = 0
    var right = array.lastIndex

    while (left < right) {
        val currentSum = array[left] + array[right]

        if (currentSum == targetSum) {
            return listOf(array[left], array[right])
        } else if (currentSum < targetSum) {
            left++
        } else {
            right--
        }
    }

    return listOf()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/insertionSort/solution1/insertionSort.kt

```
package ae.easy.insertionSort.solution1

fun insertionSort(array: MutableList<Int>): List<Int> {
    // [8,           |      5, 2, 9, 5, 6, 3]
    // sorted sublist          unsorted sublist
    // At the beginning sorted sublist has only one number, the first
element
    // Insert rest of the elements in the sorted sublist one by one
    for (i in 1 .. array.lastIndex) {
        // Track the elements, when they are continuously being swapped
inside of the sorted sublist
        var j = i

        // While we didn't reach the very beginning of the array (sorted
subarray) and the number at index j is out of order
        while (j > 0 && array[j] < array[j - 1]) {
            // Then swap the number
            swap(j, j - 1, array)
            // And j decrements to the previous index to keep track of the
number being inserted
            j -= 1
        }
    }

    return array
}

fun swap(i: Int, j: Int, array: MutableList<Int>) {
    val temp = array[i]
    array[i] = array[j]
    array[j] = temp
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/selectionSort/solution1/selectionSort.kt

```
package ae.easy.selectionSort.solution1

fun selectionSort(array: MutableList<Int>): List<Int> {
    // Initialize unsortedStartingIdx. It is the starting index of the
    // unsorted subarray
    var unsortedStartingIdx = 0

    // Loop until the unsorted subarray has only one element. At that point
    // the unsorted subarray is effectively sorted.
    while (unsortedStartingIdx < array.lastIndex) {
        // We assume that the smallest element in the unsorted subarray is
        // the first element inside it.
        var smallestElementIdx = unsortedStartingIdx

        // Loop until we consider all elements in the unsorted sublist as
        // the potential smallest element
        for (i in unsortedStartingIdx + 1 .. array.lastIndex) {
            // If current element is smaller than the current smallest
            // element, than update the smallest element index
            if (array[i] < array[smallestElementIdx]) {
                smallestElementIdx = i
            }
        }

        // We found the smallest element in the unsorted sublist. Swap it
        // with the first element of the unsorted sublist
        swap(unsortedStartingIdx, smallestElementIdx, array)

        // Smallest element in the unsorted sublist is moved to it's first
        // position. So now the unsorted sublist starts from the next position of it.
        unsortedStartingIdx += 1
    }

    return array
}

fun swap(i: Int, j: Int, array: MutableList<Int>) {
    val temp = array[i]
    array[i] = array[j]
    array[j] = temp
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/tandemBicycle/solution1/tandemBicycle.kt

```
package ae.easy.tandemBicycle.solution1

import kotlin.math.max

fun tandemBicycle(redShirtSpeeds: MutableList<Int>, blueShirtSpeeds: MutableList<Int>, fastest: Boolean): Int {
    redShirtSpeeds.sort()

    if (fastest) {
        blueShirtSpeeds.sort()
    } else {
        blueShirtSpeeds.sortWith(Comparator<Int> { i1, i2 ->
            i2.compareTo(i1)
        })
    }
}

var totalSpeed = 0

for (i in 0 .. redShirtSpeeds.lastIndex) {
    val redShirtSpeed = redShirtSpeeds[i]
    val blueShirtSpeed = blueShirtSpeeds[blueShirtSpeeds.size - i - 1]
    totalSpeed += max(redShirtSpeed, blueShirtSpeed)
}

return totalSpeed
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/palindromeCheck/solution1/palindromeCheck.kt

```
package ae.easy.palindromeCheck.solution1
```

```
fun isPalindrome(string: String): Boolean {
    var left = 0
    var right = string.lastIndex

    while (left < right) {
        if (string[left] != string[right]) {
            return false
        }

        left += 1
        right -= 1
    }

    return true
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/palindromeCheck/solution2/palindromeCheck.kt

```
package ae.easy.palindromeCheck.solution2

fun isPalindrome(string: String): Boolean {
    var left = 0
    var right = string.lastIndex

    while (left < right) {
        if (string[left] != string[right]) {
            return false
        }

        left += 1
        right -= 1
    }

    return true
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/palindromeCheck/solution3/palindromeCheck.kt

```
package ae.easy.palindromeCheck.solution3

fun isPalindrome(string: String): Boolean {
    return helper(string, 0)
}

fun helper(string: String, i: Int): Boolean {
    val j = string.lastIndex - i

    return if (j <= i) true else string[i] == string[j] && helper(string, i
+ 1)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/depthFirstSearch/solution1/depthFirstSearch.kt

```
package ae.easy.depthFirstSearch.solution1

class Node(name: String) {
    val name: String = name
    val children = mutableListOf<Node>()

    fun depthFirstSearch(): List<String> {
        return depthFirstSearch(mutableListOf())
    }

    fun depthFirstSearch(array: MutableList<String>): List<String> {
        array.add(name)

        for (child in children) {
            child.depthFirstSearch(array)
        }

        return array
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/generateDocument/solution1/generateDocument.kt

```
package ae.easy.generateDocument.solution1

fun generateDocument(characters: String, document: String): Boolean {
    val charFreq = mutableMapOf<Char, Int>()

    for (c in characters) {
        if (!charFreq.containsKey(c)) {
            charFreq[c] = 0
        }

        charFreq[c] = charFreq[c]!! + 1
    }

    for (c in document) {
        if (!charFreq.containsKey(c) || charFreq[c]!! == 0) {
            return false
        }

        charFreq[c] = charFreq[c]!! - 1
    }

    return true
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/tournamentWinner/solution1/tournamentWinner.kt

```
package ae.easy.tournamentWinner.solution1

// Simulation
fun tournamentWinner(competitions: List<List<String>>, results: List<Int>): String {
    val points = mutableMapOf<String, Int>()
    var winnerTeam = ""

    for (i in 0 .. competitions.lastIndex) {
        val (homeTeam, awayTeam) = competitions[i]
        val result = results[i]

        val currentWinner = if (result == 1) homeTeam else awayTeam

        if (!points.containsKey(currentWinner)) {
            points[currentWinner] = 0
        }
        points[currentWinner] = points[currentWinner]!! + 3

        if (winnerTeam == "" || points[currentWinner]!! > points[winnerTeam]!!) {
            winnerTeam = currentWinner
        }
    }

    return winnerTeam
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/runLengthEncoding/solution1/runLengthEncoding.kt

```
package ae.easy.runLengthEncoding.solution1
```

```
fun runLengthEncoding(string: String): String {
    val result = mutableListOf<Char>()
    var length = 1

    for (i in 1 .. string.lastIndex) {
        val currentChar = string[i]
        val prevChar = string[i - 1]

        if (currentChar != prevChar || length == 9) {
            result.add('0' + length)
            result.add(prevChar)
            length = 0
        }

        length += 1
    }

    result.add('0' + length)
    result.add(string.last())

    return String(result.toCharArray())
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/minimumWaitingTime/solution1/minimumWaitingTime.kt

```
package ae.easy.minimumWaitingTime.solution1

// [1, 2, 2, 3, 6]

// 0 + 1 + 1 + 2 + 1 + 2 + 2 + 1 + 2 + 2 + 3
// 1 -> 4
// 2 -> 3
// 2 -> 2
// 3 -> 1

// 0 + (1 * 4) = 4
// 4 + (2 * 3) = 10
// 10 + (2 * 2) = 14
// 14 + (3 * 1) = 17
// 17 + (6 * 0) = 17

fun minimumWaitingTime(queries: MutableList<Int>): Int {
    queries.sort()

    var totalWaitingTime = 0

    for (i in 0 .. queries.lastIndex) {
        val currentQuery = queries[i]
        val queriesLeft = queries.size - i - 1
        totalWaitingTime += (queriesLeft * currentQuery)
    }

    return totalWaitingTime
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/sortedSquaredArray/solution1/sortedSquaredArray.kt

```
package ae.easy.sortedSquaredArray.solution1

import kotlin.math.abs
import kotlin.math.pow

fun sortedSquaredArray(array: List<Int>): List<Int> {
    var left = 0
    var right = array.lastIndex

    val result = mutableListOf(array.size) { 0 }
    // Need to insert RIGHT TO LEFT, otherwise it will not work
    var resultIdx = array.lastIndex

    while (left <= right) {
        if (abs(array[right]) > abs(array[left])) {
            result[resultIdx] = array[right].toDouble().pow(2).toInt()
            right -= 1
        } else {
            result[resultIdx] = array[left].toDouble().pow(2).toInt()
            left += 1
        }

        resultIdx -= 1
    }

    return result
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/validateSubsequence/solution1/validateSubsequence.kt

```
package ae.easy.validateSubsequence.solution1

fun isValidSubsequence(array: List<Int>, sequence: List<Int>): Boolean {
    var aIdx = 0
    var seqIdx = 0

    while (aIdx < array.size && seqIdx < sequence.size) {
        if (array[aIdx] == sequence[seqIdx]) {
            seqIdx += 1
        }

        aIdx += 1
    }

    return seqIdx == sequence.size
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/caesarCipherEncryptor/solution1/caesarCipherEncryptor.kt

```
package ae.easy.caesarCipherEncryptor.solution1

fun caesarCipherEncryptor(string: String, key: Int): String {
    val roundedKey = key % 26

    val charArray = string.map { c ->
        val charValue = c - 'a'
        val shiftedCharValue = (charValue + roundedKey) % 26
        'a' + shiftedCharValue
    }.toCharArray()

    return String(charArray)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/findClosestValueInBST/solution1/findClosestValueInBST.kt

```
package ae.easy.findClosestValueInBST.solution1

import kotlin.math.abs

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null
}

fun findClosestValueInBst(tree: BST, target: Int): Int {
    return helper(tree, target, tree.value)
}

fun helper(tree: BST?, target: Int, closest: Int): Int {
    tree ?: return closest

    var nextClosest = closest

    if (abs(target - tree.value) < abs(target - closest)) {
        nextClosest = tree.value
    }

    if (target < tree.value) {
        nextClosest = helper(tree.left, target, nextClosest)
    } else if (target > tree.value) {
        nextClosest = helper(tree.right, target, nextClosest)
    }

    return nextClosest
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/findClosestValueInBST/solution2/findClosestValueInBST.kt

```
package ae.easy.findClosestValueInBST.solution2

import kotlin.math.abs

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null
}

fun findClosestValueInBst(tree: BST, target: Int): Int {
    var current: BST? = tree
    var closest = tree.value

    while (current != null) {
        if (abs(target - current.value) < abs(target - closest)) {
            closest = current.value
        }

        if (target < current.value) {
            current = current.left
        } else if (target > current.value) {
            current = current.right
        } else {
            break
        }
    }

    return closest
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/nonConstructibleChange/solution1/nonConstructibleChange.kt

```
package ae.easy.nonConstructibleChange.solution1

// Greedy
fun nonConstructibleChange(coins: MutableList<Int>): Int {
    // [1, 1, 2, 3, 5, 22] - 13
    coins.sort()

    var change = 0

    for (c in coins) {
        if (c > change + 1) {
            return change + 1
        }

        change += c
    }

    return change + 1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/findThreeLargestNumbers/solution1/findThreeLargestNumbers.kt

```
package ae.easy.findThreeLargestNumbers.solution1

fun findThreeLargestNumbers(array: List<Int>): List<Int> {
    val threeLargest = mutableListOf<Int>(3) { Integer.MIN_VALUE }

    for (n in array) {
        updateLargest(threeLargest, n)
    }

    return threeLargest
}

fun updateLargest(threeLargest: MutableList<Int>, num: Int) {
    if (threeLargest[2] == Integer.MIN_VALUE || num > threeLargest[2]) {
        shiftAndUpdate(threeLargest, num, 2)
    } else if (threeLargest[1] == Integer.MIN_VALUE || num > threeLargest[1]) {
        shiftAndUpdate(threeLargest, num, 1)
    } else if (threeLargest[0] == Integer.MIN_VALUE || num > threeLargest[0]) {
        shiftAndUpdate(threeLargest, num, 0)
    }
}

fun shiftAndUpdate(threeLargest: MutableList<Int>, num: Int, idx: Int) {
    for (i in 0 .. idx) {
        if (i == idx) {
            threeLargest[i] = num
        } else {
            threeLargest[i] = threeLargest[i + 1]
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/firstNonRepeatingCharacter/solution1/firstNonRepeatingCharacter.kt

```
package ae.easy.firstNonRepeatingCharacter.solution1

fun firstNonRepeatingCharacter(string: String): Int {
    for (i in string.indices) {
        var foundDuplicate = false
        for (j in string.indices) {
            if (i != j && string[i] == string[j]) {
                foundDuplicate = true
            }
        }
        if (!foundDuplicate) {
            return i
        }
    }
    return -1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/firstNonRepeatingCharacter/solution2/firstNonRepeatingCharacter.kt

```
package ae.easy.firstNonRepeatingCharacter.solution2

fun firstNonRepeatingCharacter(string: String): Int {
    val freq = string.foldMutableList(26) { 0 } { freq, c ->
        freq.also {
            freq[c - 'a'] += 1
        }
    }

    for (i in string.indices) {
        val c = string[i]
        if (freq[c - 'a'] == 1) {
            return i
        }
    }

    return -1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/easy/removeDuplicatesFromLinkedList/solution1/removeDuplicatesFromLinkedList.kt

```
package ae.easy.removeDuplicatesFromLinkedList.solution1

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun removeDuplicatesFromLinkedList(linkedList: LinkedList): LinkedList {
    var slow: LinkedList? = linkedList
    var fast: LinkedList? = linkedList

    while (fast != null) {
        while (fast != null && fast.value == slow!!.value) {
            fast = fast.next
        }

        slow!!.next = fast
        slow = fast
    }

    return linkedList
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/findLoop/solution1/findLoop.kt

```
package ae.hard.findLoop.solution1
```

```
open class LinkedList(value: Int) {  
    var value = value  
    var next: LinkedList? = null  
}  
  
fun findLoop(head: LinkedList?): LinkedList? {  
    var first = head?.next  
    var second = head?.next?.next  
  
    while (first != second) {  
        first = first?.next  
        second = second?.next?.next  
    }  
  
    first = head  
  
    while (first != second) {  
        first = first?.next  
        second = second?.next  
    }  
  
    return first  
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/heapSort/solution1/heapSort.kt

```
package ae.hard.heapSort.solution1

fun heapSort(array: MutableList<Int>): List<Int> {
    buildHeap(array)
    for (i in array.lastIndex downTo 1) {
        swap(0, i, array)
        siftDown(0, i - 1, array)
    }
    return array
}

fun swap(i: Int, j: Int, array: MutableList<Int>) {
    array[i] = array[j].also {
        array[j] = array[i]
    }
}

fun buildHeap(array: MutableList<Int>) {
    val firstParentIdx = Math.floor((array.size - 1.0) / 2.0).toInt()
    for (i in firstParentIdx downTo 0) {
        siftDown(i, array.size - 1, array)
    }
}

fun siftDown(startIdx: Int, endIdx: Int, array: MutableList<Int>) {
    var runningStartIdx = startIdx
    var firstChildIdx = runningStartIdx * 2 + 1
    while (firstChildIdx <= endIdx) {
        val secondChildIdx = runningStartIdx * 2 + 2

        val idxToSwap = if (secondChildIdx <= endIdx && array[secondChildIdx] > array[firstChildIdx]) {
            secondChildIdx
        } else firstChildIdx

        if (array[idxToSwap] > array[runningStartIdx]) {
            swap(idxToSwap, runningStartIdx, array)
            runningStartIdx = idxToSwap
            firstChildIdx = runningStartIdx * 2 + 1
        } else {
            return
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/sameBSTs/solution1/sameBSTs.kt

```
package ae.hard.sameBSTs.solution1

fun sameBsts(arrayOne: List<Int>, arrayTwo: List<Int>): Boolean {
    if (arrayOne.size != arrayTwo.size) return false

    if (arrayOne.isEmpty() && arrayTwo.isEmpty()) return true

    if (arrayOne[0] != arrayTwo[0]) return false

    val leftOne = getSmaller(arrayOne)
    val rightOne = getGreaterThanOrEqualTo(arrayOne)

    val leftTwo = getSmaller(arrayTwo)
    val rightTwo = getGreaterThanOrEqualTo(arrayTwo)

    return sameBsts(leftOne, leftTwo) && sameBsts(rightOne, rightTwo)
}

fun getSmaller(array: List<Int>): List<Int> {
    val rootValue = array[0]
    return array.subList(1, array.size).filter { it < rootValue }
}

fun getGreaterThanOrEqualTo(array: List<Int>): List<Int> {
    val rootValue = array[0]
    return array.subList(1, array.size).filter { it >= rootValue }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/sameBSTs/solution2/sameBSTs.kt

```
package ae.hard.sameBSTs.solution2

fun sameBsts(arrayOne: List<Int>, arrayTwo: List<Int>): Boolean {
    return areSameBsts(arrayOne, arrayTwo, 0, 0, Int.MIN_VALUE, Int.
MAX_VALUE)
}

fun areSameBsts(arrayOne: List<Int>, arrayTwo: List<Int>, rootIdxOne: Int,
rootIdxTwo: Int, minVal: Int, maxVal: Int): Boolean {
    if (rootIdxOne == -1 || rootIdxTwo == -1) {
        return rootIdxOne == rootIdxTwo
    }

    if (arrayOne[rootIdxOne] != arrayTwo[rootIdxTwo]) {
        return false
    }

    val leftIdxOne = getIdxOfFirstSmaller(arrayOne, rootIdxOne, minVal)
    val leftIdxTwo = getIdxOfFirstSmaller(arrayTwo, rootIdxTwo, minVal)

    val rightIdxOne = getIdxOfFirstGreaterThanOrEqual(arrayOne, rootIdxOne,
, maxVal)
    val rightIdxTwo = getIdxOfFirstGreaterThanOrEqual(arrayTwo, rootIdxTwo,
, maxVal)

    val currentVal = arrayOne[rootIdxOne]
    val areLeftSame = areSameBsts(arrayOne, arrayTwo, leftIdxOne,
leftIdxTwo, minVal, currentVal)
    val areRightSame = areSameBsts(arrayOne, arrayTwo, rightIdxOne,
rightIdxTwo, currentVal, maxVal)

    return areLeftSame && areRightSame
}

fun getIdxOfFirstSmaller(array: List<Int>, rootIdx: Int, minVal: Int): Int
{
    for (i in rootIdx + 1 .. array.lastIndex) {
        if (array[i] < array[rootIdx] && array[i] >= minVal) {
            return i
        }
    }
}

return -1
}

fun getIdxOfFirstGreaterThanOrEqual(array: List<Int>, rootIdx: Int, maxVal
: Int): Int {
    for (i in rootIdx + 1 .. array.lastIndex) {
        if (array[i] >= array[rootIdx] && array[i] < maxVal) {
            return i
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/sameBSTs/solution2/sameBSTs.kt

```
    return -1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/quickSort/solution1/quickSort.kt

```
package ae.hard.quickSort.solution1

fun quickSort(array: MutableList<Int>): List<Int> {
    quickSortHelper(array, 0, array.lastIndex)
    return array
}

fun quickSortHelper(array: MutableList<Int>, start: Int, end: Int) {
    // If start meets end, then return
    if (start >= end) {
        return
    }

    // Pick start value as pivot
    var pivot = start
    // Pick second value as left
    var left = start + 1
    // Pick end value as right
    var right = end

    // Until left meets right
    // All the elements smaller than pivot should come before it.
    // All elements greater than pivot should come after it.
    // If this is not the case then we need to swap.
    while (left <= right) {

        // If array element at left is greater than element at pivot
        // And if element at right is less than element at pivot
        // Then we did find two elements which are out of order
        // We will swap them
        if (array[left] > array[pivot] && array[right] < array[pivot]) {
            swap(left, right, array)
        }

        // If element at left is less than or equal to element at pivot
        // Then it's in correct position
        // Otherwise we will freeze left until we find a right which is
        // also in incorrect position
        if (array[left] <= array[pivot]) {
            left += 1
        }

        // If element at right is greater than or equal to element at pivot
        // Then it's in correct position
        // Otherwise we will freeze right until we find a left which is
        // also in incorrect position
        if (array[right] >= array[pivot]) {
            right -= 1
        }
    }

    // Swap element at pivot and at right
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/quickSort/solution1/quickSort.kt

```
// Now we can say that element at right is in it's correct sorted
position
swap(pivot, right, array)

// New pivot is right, so which half of the array is smaller?
// We need to divide the array around it
// Is it right - 1 - start or end - (right + 1)
val isLeftSmaller = right - 1 - start < end - (right + 1)

// If left part is smaller
// then call quick sort recursively on the left part first and then on
the right part
// Otherwise call quick sort recursively on the right part first and then on
the left part
if (isLeftSmaller) {
    quickSortHelper(array, start, right - 1)
    quickSortHelper(array, right + 1, end)
} else {
    quickSortHelper(array, right + 1, end)
    quickSortHelper(array, start, right - 1)
}
}

fun swap(leftIdx: Int, rightIdx: Int, array: MutableList<Int>) {
    array[leftIdx] = array[rightIdx].also {
        array[rightIdx] = array[leftIdx]
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/radixSort/solution1/radixSort.kt

```
package ae.hard.radixSort.solution1

import kotlin.math.pow

// O(d * (n + b)) Time | O(n + b) space
fun radixSort(array: MutableList<Int>): MutableList<Int> {

    if (array.isEmpty()) return array

    val max = array.maxOrNull()!!
    var position = 0

    // Position is the 10's place e.g.
    // For 210, 0 has position 0, 1 has 1 and 2 has 2
    while (max / 10.toDouble().pow(position) > 0) {
        countingSort(array, position)
        position += 1
    }

    return array
}

fun countingSort(array: MutableList<Int>, position: Int) {
    val counts = MutableList(10) { 0 }
    val sorted = MutableList(array.size) { 0 }

    val divider = 10.toDouble().pow(position).toInt()

    for (current in array) {
        val digit = (current / divider) % 10
        counts[digit] += 1
    }

    // We need to modify counts array counts[i] += counts[i - 1]
    // this way to find the occurrence of last position of a digit
    // As example [ 0, 0, 3, 5, 7, 7, 8, 9, 9] means that
    // First time we see a 2 it will be at position 3 (index 2)
    // Second time we see a 2 it will be at position 2 (index 1)
    // Third time we see a 2 it will be at position 1 (index 0)
    for (i in 1 until 10) {
        counts[i] += counts[i - 1]
    }

    // We need to iterate backward while inserting elements
    // Is to keep the previous ordering of the
    // partial sorted array we've done before.
    // e.g when sorting by position 1
    // we need to keep the sorting done by position 0.
    for (i in array.lastIndex downTo 0) {
        val currentNum = array[i]
        val digit = (currentNum / divider) % 10
        counts[digit] -= 1
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/radixSort/solution1/radixSort.kt

```
    val indexToInsert = counts[digit]
    sorted[indexToInsert] = currentNum
}

for (i in 0 .. array.lastIndex) {
    array[i] = sorted[i]
}
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/waterArea/solution1/waterArea.kt

```
package ae.hard.waterArea.solution1

import kotlin.math.max
import kotlin.math.min

fun waterArea(heights: List<Int>): Int {
    val maxes = mutableListOf(heights.size) { 0 }
    var area = 0

    var leftMax = 0
    for (i in 0 .. heights.lastIndex) {
        maxes[i] = leftMax
        val height = heights[i]
        leftMax = max(leftMax, height)
    }

    var rightMax = 0
    for (i in heights.lastIndex downTo 0) {
        val minHeight = min(maxes[i], rightMax)
        val height = heights[i]

        if (height < minHeight) {
            maxes[i] = minHeight - height
        } else {
            maxes[i] = 0
        }

        area += maxes[i]
        rightMax = max(rightMax, height)
    }

    return area
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/minRewards/solution1/minRewards.kt

```
package ae.hard.minRewards.solution1

import kotlin.math.max

fun minRewards(scores: List<Int>): Int {
    val rewards = MutableList(scores.size) { 1 }

    for (i in 1 until scores.size) {

        var j = i - 1

        if (scores[i] > scores[j]) {
            rewards[i] = rewards[j] + 1
        } else {
            while (j >= 0 && scores[j] > scores[j + 1]) {
                rewards[j] = max(rewards[j], rewards[j + 1] + 1)
                j -= 1
            }
        }
    }

    return rewards.sum()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/minRewards/solution2/minRewards.kt

```
package ae.hard.minRewards.solution2

import kotlin.math.max

fun minRewards(scores: List<Int>): Int {
    val rewards = MutableList(scores.size) { 1 }

    for (valleyIdx in findValleyIdxs(scores)) {
        expandAroundValley(valleyIdx, scores, rewards)
    }

    return rewards.sum()
}

fun findValleyIdxs(array: List<Int>): List<Int> {
    if (array.size == 1) return listOf(0)

    val valleyIdxs = mutableListOf<Int>()

    for (i in array.indices) {
        if (i == 0 && array[i] < array[i + 1]) {
            valleyIdxs.add(i)
        } else if (i == array.lastIndex && array[i] < array[i - 1]) {
            valleyIdxs.add(i)
        } else if (i == 0 || i == array.lastIndex) {
            continue
        } else if (array[i] < array[i - 1] && array[i] < array[i + 1]) {
            valleyIdxs.add(i)
        }
    }

    return valleyIdxs
}

fun expandAroundValley(valleyIdx: Int, scores: List<Int>, rewards: MutableList<Int>) {
    var leftIdx = valleyIdx - 1
    while (leftIdx >= 0 && scores[leftIdx] > scores[leftIdx + 1]) {
        rewards[leftIdx] = max(rewards[leftIdx], rewards[leftIdx + 1] + 1)
        leftIdx -= 1
    }

    var rightIdx = valleyIdx + 1
    while (rightIdx < scores.size && scores[rightIdx] > scores[rightIdx - 1])
    {
        rewards[rightIdx] = rewards[rightIdx - 1] + 1
        rightIdx += 1
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/minRewards/solution3/minRewards.kt

```
package ae.hard.minRewards.solution3

import kotlin.math.max

fun minRewards(scores: List<Int>): Int {

    val rewards = mutableListOf(scores.size) { 1 }

    for (i in 1 until scores.size) {
        if (scores[i] > scores[i - 1]) {
            rewards[i] = rewards[i - 1] + 1
        }
    }

    for (i in scores.lastIndex - 1 downTo 0) {
        if (scores[i] > scores[i + 1]) {
            rewards[i] = max(rewards[i], rewards[i + 1] + 1)
        }
    }

    return rewards.sum()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/boggleBoard/solution1/boggleBoard.kt

```
package ae.hard.boggleBoard.solution1

const val DELIMITER = '*'

fun boggleBoard(board: List<List<Char>>, words: List<String>): List<String>
> {
    val root = populateSuffixTrie(words)

    val result = mutableSetOf<String>()
    val visited = MutableList(board.size) { MutableList(board[0].size) { false } }

    for (r in board.indices) {
        for (c in board[0].indices) {
            dfsVisitNode(r, c, root, board, visited, result)
        }
    }

    return result.toList()
}

class TrieNode {
    val children: MutableMap<Char, TrieNode> = mutableMapOf()
    var word: String = ""
}

fun populateSuffixTrie(words: List<String>): TrieNode {
    val root = TrieNode()

    for (word in words) {
        var currentNode = root

        for (c in word) {
            if (!currentNode.children.containsKey(c)) {
                currentNode.children[c] = TrieNode()
            }

            currentNode = currentNode.children[c]!!
        }

        val delimiterNode = TrieNode().apply {
            this.word = word
        }
        currentNode.children[DELIMITER] = delimiterNode
    }

    return root
}

fun dfsVisitNode(r: Int, c: Int, currentNode: TrieNode, board: List<List<Char>>, visited: MutableList<MutableList<Boolean>>, result: MutableSet<String>) {
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/boggleBoard/solution1/boggleBoard.kt

```
if (visited[r][c]) return

val letter = board[r][c]

if (!currentNode.children.containsKey(letter)) return

visited[r][c] = true

val nextNode = currentNode.children[letter]!!

if (nextNode.children.containsKey(DELIMITER)) {
    result.add(nextNode.children[DELIMITER]!! .word)
}

val neighbors = getNeighbors(r, c, board)

for ((neighborRow, neighborCol) in neighbors) {
    dfsVisitNode(neighborRow, neighborCol, nextNode, board, visited,
result)
}

visited[r][c] = false
}

fun getNeighbors(r: Int, c: Int, board: List<List<Char>>): List<Pair<Int,
Int>> {
    val rows = board.size
    val cols = board[0].size

    return listOf(
        r - 1 to c + 1 ,
        r to c + 1,
        r + 1 to c + 1,
        r + 1 to c ,
        r + 1 to c - 1,
        r to c - 1,
        r - 1 to c - 1,
        r - 1 to c
    ).filter { (row, col) ->
        row in 0 until rows && col in 0 until cols
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/numbersInPi/solution1/numbersInPi.kt

```
package ae.hard.numbersInPi.solution1

import kotlin.math.min

fun numbersInPi(pi: String, numbers: List<String>): Int {
    val numbersSet = mutableSetOf<String>()
    for (number in numbers) {
        numbersSet.add(number)
    }

    val cache = mutableMapOf<Int, Int>()
    val minSpaces = getMinSpaces(pi, numbersSet, cache, 0)

    return if (minSpaces == Integer.MAX_VALUE) -1 else minSpaces
}

fun getMinSpaces(pi: String, numbersSet: Set<String>, cache: MutableMap<Int, Int>, idx: Int): Int {
    if (idx == pi.length) {
        return -1
    }

    if (cache.containsKey(idx)) {
        return cache[idx]!!
    }

    var minSpaces = Integer.MAX_VALUE

    for (i in idx .. pi.lastIndex) {
        // Substring inclusive both idx and i
        val prefix = pi.substring(idx, i + 1)

        if (numbersSet.contains(prefix)) {
            val minSpacesSuffix = getMinSpaces(pi, numbersSet, cache, i + 1)
        }
        minSpaces = min(
            minSpaces,
            if (minSpacesSuffix == Integer.MAX_VALUE) minSpacesSuffix
        )
    }

    cache[idx] = minSpaces
    return cache[idx]!!
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/numbersInPi/solution2/numbersInPi.kt

```
package ae.hard.numbersInPi.solution2

import kotlin.math.min

fun numbersInPi(pi: String, numbers: List<String>): Int {
    val numbersSet = mutableSetOf<String>()
    for (number in numbers) {
        numbersSet.add(number)
    }

    val cache = mutableMapOf<Int, Int>()

    for (idx in pi.lastIndex downTo 0) {
        val minSpaces = getMinSpaces(pi, numbersSet, cache, idx)
    }

    return if (cache[0]!! == Integer.MAX_VALUE) -1 else cache[0]!!
}

fun getMinSpaces(pi: String, numbersSet: Set<String>, cache: MutableMap<Int, Int>, idx: Int): Int {
    if (idx == pi.length) {
        return -1
    }

    if (cache.containsKey(idx)) {
        return cache[idx]!!
    }

    var minSpaces = Integer.MAX_VALUE

    for (i in idx .. pi.lastIndex) {
        // Substring inclusive both idx and i
        val prefix = pi.substring(idx, i + 1)

        if (numbersSet.contains(prefix)) {
            val minSpacesSuffix = getMinSpaces(pi, numbersSet, cache, i + 1)
        }
        minSpaces = min(
            minSpaces,
            if (minSpacesSuffix == Integer.MAX_VALUE) minSpacesSuffix
        )
    }

    cache[idx] = minSpaces
    return cache[idx]!!
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/quickSelect/solution1/quickSelect.kt

```
package ae.hard.quickSelect.solution1

fun quickselect(array: MutableList<Int>, k: Int): Int {
    val position = k - 1
    return quickselectHelper(array, 0, array.lastIndex, position)
}

fun quickselectHelper(array: MutableList<Int>, startIdx: Int, endIdx: Int,
position: Int): Int {
    var runningStartIdx = startIdx
    var runningEndIdx = endIdx

    while (true) {
        if (runningStartIdx > runningEndIdx) {
            throw Exception("Illegal state")
        }

        // Pick start value as pivot
        val pivotIdx = runningStartIdx
        // Pick second value as left
        var leftIdx = runningStartIdx + 1
        // Pick end value as right
        var rightIdx = runningEndIdx

        // Until left meets right
        // All the elements smaller than pivot should come before it.
        // All elements greater than pivot should come after it.
        // If this is not the case then we need to swap.
        while (leftIdx <= rightIdx) {

            // If array element at left is greater than element at pivot
            // And if element at right is less than element at pivot
            // Then we did find two elements which are out of order
            // We will swap them
            if (array[leftIdx] > array[pivotIdx] && array[rightIdx] < array
[pivotIdx]) {
                swap(leftIdx, rightIdx, array)
            }

            // If element at left is less than or equal to element at pivot
            // Then it's in correct position
            // Otherwise we will freeze left until we find a right which is
also in incorrect position
            if (array[leftIdx] <= array[pivotIdx]) {
                leftIdx++
            }

            // If element at right is greater than or equal to element at
pivot
            // Then it's in correct position
            // Otherwise we will freeze right until we find a left which is
also in incorrect position
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/quickSelect/solution1/quickSelect.kt

```
    if (array[rightIdx] >= array[pivotIdx]) {
        rightIdx--
    }
}

// Swap element at pivot and at right
// Now we can say that element at right is in it's correct sorted
position
swap(pivotIdx, rightIdx, array)

// element in right is the kth smallest integer
if (rightIdx === position) {
    return array[rightIdx]

    // rightIdx is in it's correct sorted position
    // ALSO All the number before rightIdx is smaller than it and
    all the number after it is greater than it
    // If rightIdx is greater then position this means kth smallest
    number should be smaller and to the left of rightIdx
    // So look at the left
} else if (rightIdx > position) {
    runningEndIdx = rightIdx - 1
    // Otherwise element at rightIdx is smaller than kth smallest
number
    // So look at right of rightIdx
} else {
    runningStartIdx = rightIdx + 1
}
}

fun swap(leftIdx: Int, rightIdx: Int, array: MutableList<Int>) {
    array[leftIdx] = array[rightIdx].also {
        array[rightIdx] = array[leftIdx]
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/shortenPath/solution1/shortenPath.kt

```
package ae.hard.shortenPath.solution1

fun isImportantToken(token: String) =
    token.isNotEmpty() && token != "."

fun shortenPath(path: String): String {
    val filteredTokens = path.split("/").filter(::isImportantToken)

    val stack = mutableListOf<String>()

    if (path.startsWith('/')) {
        stack.add("")
    }

    filteredTokens.forEach { token ->
        if (token == "..") {
            if (stack.isEmpty() || stack.last() == "..") {
                stack.add(token)
            } else if (stack.last() != "") {
                stack.removeAt(stack.lastIndex)
            }
        } else {
            stack.add(token)
        }
    }

    if (stack.size == 1 && stack[0] == "") {
        return "/"
    }

    return stack.joinToString("/")
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/solveSudoku/solution1/solveSudoku.kt

```
package ae.hard.solveSudoku.solution1

fun solveSudoku(board: MutableList<MutableList<Int>>): MutableList<
MutableList<Int>> {
    // Start solving sudoku board recursively and change it in-place.
    solvePartialSudoku(0, 0, board)
    return board
}

// Solve sudoku board per-position recursively.
fun solvePartialSudoku(row: Int, col: Int, board: MutableList<MutableList<
Int>>): Boolean {
    var currentRow = row
    var currentCol = col

    // Column value get's incremented to make next recursive call.
    // If the column value gets passed the maximum column value then wrap
    column to 0.
    // Then increment the row to the next row.
    if (currentCol == board[0].size) {
        currentCol = 0
        currentRow += 1
    }

    // If the current row gets passed the maximum row value then the sudoku
    board is fully solved.
    // Return true in that case.
    if (currentRow == board.size) {
        return true
    }

    // If the board's current cell value is 0 (e.g the cell is unoccupied)
    // Then try to solve the cell.
    if (board[currentRow][currentCol] == 0) {
        return tryDigitsAtPosition(currentRow, currentCol, board)
    }

    // Otherwise current cell has value (it is not unoccupied).
    // Solve recursively for the next column.
    return solvePartialSudoku(currentRow, currentCol + 1, board)
}

fun tryDigitsAtPosition(row: Int, col: Int, board: MutableList<MutableList<
Int>>): Boolean {
    // Try placing each digit between 1-9.
    for (digit in 1 .. 9) {
        // If this digit is valid in the position marked by row and col
        if (isValidAtPosition(digit, row, col, board)) {
            // Place the digit in the cell pointed by row and col.
            board[row][col] = digit

            // If we can solve the next column recursively in a valid way
        }
    }
}
```

```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/solveSudoku/solution1/solveSudoku.kt
then this position is valid.
    // Return true in that case.
    if (solvePartialSudoku(row, col + 1, board)) {
        return true
    }
}

// We reach here if no digit is valid in the position of row, col
// OR we could not solve the next column (col + 1) recursively.

// We mark the position marked with row, col as unoccupied and we
return false.
board[row][col] = 0
return false
}

// This function checks if a value is valid in a row, col position
fun isValidAtPosition(digit: Int, row: Int, col: Int, board: MutableList<
MutableList<Int>>): Boolean {
    // If the value is not present in currentRow then the value is valid in
current row.
    val isValidInRow = !board[row].contains(digit)

    // If the value is not present in currentCol, then the value is valid
in current col.
    val isValidInCol = board.fold(true) { isValid, currentRow ->
        isValid && currentRow[col] != digit
    }

    // If row or col is invalid, then just return false.
    if (!isValidInRow || !isValidInCol) {
        return false
    }

    // Get the 3x3 grid number of the passed row and column.
    // Example if passed row, col is 5, 7,
    // rowGridNumber = 1
    // colGridNumber = 2
    val rowGridNumber = row / 3
    val colGridNumber = col / 3

    // We only navigate in the current 3x3 grid.
    for (r in 0 until 3) {
        // We get the row index within the grid, gridNumber * 3 + r.
        val rowGridIdx = rowGridNumber * 3 + r

        for (c in 0 until 3) {
            // We get the col index within the grid, gridNumber * 3 + c.
            val colGridIdx = colGridNumber * 3 + c

            // If in the current position of the 3x3 grid, we find the

```

```
File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/solveSudoku/solution1/solveSudoku.kt
value, then it is invalid.
    if (board[rowGridIdx][colGridIdx] == digit) {
        return false
    }
}

// We passed all above sanity check, so the value is valid.
return true
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/diskStacking/solution1/diskStacking.kt

```
package ae.hard.diskStacking.solution1

fun diskStacking(disks: List<List<Int>>): List<List<Int>> {
    val sortedDisks = disks.sortedWith(Comparator<List<Int>> { disk1, disk2
        ->
            disk1[2].compareTo(disk2[2])
    })

    val heights = sortedDisks.map { it[2] }.toMutableList()

    val sequences = MutableList(disks.size) { -1 }
    var maxHeightIdx = 0

    for (i in 1 .. disks.lastIndex) {
        val currentDisk = sortedDisks[i]

        for (j in 0 until i) {
            val otherDisk = sortedDisks[j]

            if (currentDisk[0] > otherDisk[0] && currentDisk[1] > otherDisk[1] && currentDisk[2] > otherDisk[2]) {
                if (heights[j] + currentDisk[2] > heights[i]) {
                    heights[i] = heights[j] + currentDisk[2]
                    sequences[i] = j
                }
            }
        }

        if (heights[i] > heights[maxHeightIdx]) {
            maxHeightIdx = i
        }
    }

    return buildSequences(sequences, sortedDisks, maxHeightIdx)
}

fun buildSequences(sequences: List<Int>, disks: List<List<Int>>, maxHeightIdx: Int): List<List<Int>> {
    var currentIdx = maxHeightIdx
    val result = mutableListOf<List<Int>>()

    while (currentIdx != -1) {
        result.add(disks[currentIdx])
        currentIdx = sequences[currentIdx]
    }

    return result.reversed()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/largestRange/solution1/largestRange.kt

```
package ae.hard.largestRange.solution1

fun largestRange(array: List<Int>): Pair<Int, Int> {
    var bestRange = array[0] to array[0]
    var longestLength = 0
    val nums = array.associateWith { true }.toMutableMap()

    for (num in array) {
        if (!nums[num]!!) {
            continue
        }

        nums[num] = false
        var currentLength = 1
        var left = num - 1
        var right = num + 1

        while (nums.containsKey(left)) {
            nums[left] = false
            currentLength += 1
            left -= 1
        }

        while (nums.containsKey(right)) {
            nums[right] = false
            currentLength += 1
            right += 1
        }

        if (currentLength > longestLength) {
            longestLength = currentLength
            bestRange = left + 1 to right - 1
        }
    }

    return bestRange
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/subarraySort/solution1/subarraySort.kt

```
package ae.hard.subarraySort.solution1

import kotlin.math.min
import kotlin.math.max

// Find the smallest and largest numbers that are out of order in the input
// array.
// Once the smallest and largest out of order numbers are found
// we need to find their final sorted positions.
// This will be the extremities of the smallest subarray that needs to be
sorted.

fun subarraySort(array: List<Int>): List<Int> {
    var minOutOfOrder = Int.MAX_VALUE
    var maxOutOfOrder = Int.MIN_VALUE

    for (i in 0 .. array.lastIndex) {
        if (isOutOfOrder(i, array)) {
            minOutOfOrder = min(minOutOfOrder, array[i])
            maxOutOfOrder = max(maxOutOfOrder, array[i])
        }
    }

    if (minOutOfOrder == Int.MAX_VALUE) {
        return listOf(-1, -1)
    }

    var subarrayLeftIdx = 0
    while (array[subarrayLeftIdx] <= minOutOfOrder) {
        subarrayLeftIdx += 1
    }

    var subarrayRightIdx = array.lastIndex
    while (array[subarrayRightIdx] >= maxOutOfOrder) {
        subarrayRightIdx -= 1
    }

    return listOf(subarrayLeftIdx, subarrayRightIdx)
}

fun isOutOfOrder(idx: Int, array: List<Int>): Boolean {
    return when (idx) {
        0 -> array[0] > array[1]
        array.lastIndex -> array[array.lastIndex] < array[array.lastIndex
- 1]
        else -> array[idx] < array[idx - 1] || array[idx] > array[idx + 1]
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/fourNumberSum/solution1/fourNumberSum.kt

```
package ae.hard.fourNumberSum.solution1

// To avoid duplicates we need following mechanism.
// Loop through each element i in the array
// For each item j after i compute sum = array[i] + array[j]
// If diff = target - sum is found in the map then build quadruplets with
// array[i], array[j] and items found in the map
// If diff is not found then DO NOT add the pair in the map.
// For each element k from 0 until i compute sum = array[i] + array[k]
// Add { array[k], array[i] } against sum into the map

fun fourNumberSum(array: MutableList<Int>, targetSum: Int): List<List<Int>>
>> {
    val allPairSums = mutableMapOf<Int, MutableList<List<Int>>>()
    val quadruplets = mutableListOf<List<Int>>()

    for (i in 1 until array.lastIndex) {

        for (j in i + 1 .. array.lastIndex) {
            val currentSum = array[i] + array[j]
            val difference = targetSum - currentSum

            if (allPairSums.contains(difference)) {
                for (pair in allPairSums[difference]!!) {
                    val quadruplet = pair.plus(listOf(array[i], array[j]))
                    quadruplets.add(quadruplet)
                }
            }
        }

        for (k in 0 until i) {
            val currentSum = array[i] + array[k]

            if (allPairSums.contains(currentSum)) {
                allPairSums[currentSum]!!.add(listOf(array[k], array[i]))
            } else {
                allPairSums[currentSum] = mutableListOf(listOf(array[k],
array[i]))
            }
        }
    }

    return quadruplets
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/laptopRentals/solution1/laptopRentals.kt

```
package ae.hard.laptopRentals.solution1

import java.util.PriorityQueue
import kotlin.Comparator

fun laptopRentals(times: List<List<Int>>): Int {

    // Sort the time intervals according to their start time.
    val sortedTimes = times.sortedWith(Comparator<List<Int>> { time1, time2
        ->
            time1[0].compareTo(time2[0])
    })

    // Create a heap as the laptop pool
    // Min Heap
    val pool = PriorityQueue<Int>()

    // For each of the sorted intervals
    for (time in sortedTimes) {

        // If pool size is greater than 0
        // and pool's top end-time (earliest end time) is less than or
        equal to the next rental's start-time
        // Then we can use the same laptop.
        if (pool.isNotEmpty() && pool.peek() <= time[0]) {
            // So remove the end time from the pool.
            pool.poll()
        }

        // Regardless of anything specific we need to insert current
        interval's end time into the pool
        // Depending on the previous if statement it can be reassignment of
        an existing laptop or a new laptop
        // Existing laptop when it's previous rental ended before the start
        of the current rental.
        pool.add(time[1])
    }

    // Finally the pool size is the required number of laptops.
    return pool.size
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/laptopRentals/solution2/laptopRentals.kt

```
package ae.hard.laptopRentals.solution2

fun laptopRentals(times: List<List<Int>>): Int {
    // We don't need start and end times paired up here.
    // We need sorted versions of start and end times
    // to know which rental is starting next and which rental is ending
    // next separately
    val startTimes = times.map{ it[0] }.sorted()
    val endTimes = times.map { it[1] }.sorted()

    // Number of laptops for rental
    var laptops = 0

    // pointers to the sorted arrays startTimes and endTimes
    var startIdx = 0
    var endIdx = 0

    // We loop until startIdx is in bounds
    // endIdx progresses (or stands) along with startIdx, so we don't need
    // to check it
    while (startIdx < startTimes.size) {
        // If the earliest start time starts at or after the earliest end
        // time
        // Then we have a laptop freed up
        if (startTimes[startIdx] >= endTimes[endIdx]) {
            // We decrement laptops as a count of the freed up laptop
            laptops -= 1
            // We will consider the next ending time on next iteration
            endIdx += 1
        }

        // Anyway we need to allocate current rental, so we always
        // increment laptops count
        laptops += 1
        // We increment index for start indexes
        startIdx += 1
    }

    // We return laptop count
    return laptops
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/patternMatcher/solution1/patternMatcher.kt

```
package ae.hard.patternMatcher.solution1

import kotlin.math.floor

fun patternMatcher(pattern: String, string: String): List<String> {
    if (pattern.length > string.length) {
        return listOf()
    }

    val transformedPattern = transformPattern(pattern)
    val isFlipped = transformedPattern[0] != pattern[0]

    val charCounts = mutableMapOf('x' to 0, 'y' to 0)
    val firstYPosition = countCharsAndGetFirstYPosition(transformedPattern,
        charCounts)

    if (charCounts['y']!! > 0) {
        for (xSize in 1 .. string.lastIndex) {
            val ySizeDouble = (string.length - charCounts['x']!! * xSize).
            toDouble() / charCounts['y']!!
            if (ySizeDouble <= 0 || floor(ySizeDouble) != ySizeDouble)
                continue
            val ySize = ySizeDouble.toInt()
            val yStartingIdx = firstYPosition!! * xSize

            val x = string.substring(0, xSize)
            val y = string.substring(yStartingIdx, yStartingIdx + ySize)

            val potentialMatch = transformedPattern.map { c ->
                if (c == 'x') x else y
            }.joinToString("")

            if (potentialMatch == string) {
                return if (isFlipped) listOf(y, x) else listOf(x, y)
            }
        }
    } else {
        val xSizeDouble = string.length.toDouble() / charCounts['x']!!
        if (floor(xSizeDouble) != xSizeDouble) return listOf()
        val xSize = xSizeDouble.toInt()
        val x = string.substring(0, xSize)
        val potentialMatch = transformedPattern.map { x }.joinToString("")

        if (potentialMatch == string) {
            return if (isFlipped) listOf("", x) else listOf(x, "")
        }
    }
}

return listOf()
}

fun transformPattern(pattern: String): CharArray {
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/patternMatcher/solution1/patternMatcher.kt

```
if (pattern[0] == 'x') return pattern.toCharArray()

return pattern.map { c ->
    if (c == 'y') 'x' else 'y'
}.toCharArray()
}

fun countCharsAndGetFirstYPosition(pattern: CharArray, charCounts:
MutableMap<Char, Int>): Int? {
    var firstYPosition: Int? = null

    pattern.forEachIndexed { idx, c ->
        charCounts[c] = charCounts[c]!! + 1
        if (c == 'y' && firstYPosition == null) {
            firstYPosition = idx
        }
    }

    return firstYPosition
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/searchForRange/solution1/searchForRange.kt

```
package ae.hard.searchForRange.solution1

fun searchForRange(array: List<Int>, target: Int): List<Int> {
    val range = mutableListOf(-1, -1)
    alteredBinarySearch(array, target, 0, array.lastIndex, range, true)
    alteredBinarySearch(array, target, 0, array.lastIndex, range, false)
    return range
}

fun alteredBinarySearch(array: List<Int>, target: Int, left: Int, right: Int, range: MutableList<Int>, goLeft: Boolean) {
    if (left > right) return

    val mid = (left + right) / 2

    if (target < array[mid]) {
        alteredBinarySearch(array, target, left, mid - 1, range, goLeft)
    } else if (target > array[mid]) {
        alteredBinarySearch(array, target, mid + 1, right, range, goLeft)
    } else {
        if (goLeft) {
            if (mid == 0 || array[mid - 1] != target) {
                range[0] = mid
            } else {
                alteredBinarySearch(array, target, left, mid - 1, range, goLeft)
            }
        } else {
            if (mid == array.lastIndex || array[mid + 1] != target) {
                range[1] = mid
            } else {
                alteredBinarySearch(array, target, mid + 1, right, range, goLeft)
            }
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/searchForRange/solution2/searchForRange.kt

```
package ae.hard.searchForRange.solution2

fun searchForRange(array: List<Int>, target: Int): List<Int> {
    val range = mutableListOf(-1, -1)
    alteredBinarySearch(array, target, 0, array.lastIndex, range, true)
    alteredBinarySearch(array, target, 0, array.lastIndex, range, false)
    return range
}

fun alteredBinarySearch(array: List<Int>, target: Int, left: Int, right: Int, range: MutableList<Int>, goLeft: Boolean) {
    var leftIdx = left
    var rightIdx = right

    while (leftIdx <= rightIdx) {
        val mid = (leftIdx + rightIdx) / 2

        if (target < array[mid]) {
            rightIdx = mid - 1
        } else if (target > array[mid]) {
            leftIdx = mid + 1
        } else {
            if (goLeft) {
                if (mid == 0 || array[mid - 1] != target) {
                    range[0] = mid
                    return
                } else {
                    rightIdx = mid - 1
                }
            } else {
                if (mid == array.lastIndex || array[mid + 1] != target) {
                    range[1] = mid
                    return
                } else {
                    leftIdx = mid + 1
                }
            }
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/zigzagTraverse/solution1/zigzagTraverse.kt

```
package ae.hard.zigzagTraverse.solution1

fun zigzagTraverse(array: List<List<Int>>): List<Int> {
    val lastRow = array.lastIndex
    val lastCol = array[0].lastIndex

    val result = mutableListOf<Int>()
    var goingDown = true

    var row = 0
    var col = 0

    while (!isOutOfBounds(row, col, lastRow, lastCol)) {
        result.add(array[row][col])

        if (goingDown) {

            if (col == 0 || row == lastRow) {
                goingDown = false

                if (row == lastRow) {
                    col += 1
                } else {
                    row += 1
                }
            } else {
                row += 1
                col -= 1
            }
        } else {

            if (row == 0 || col == lastCol) {
                goingDown = true

                if (col == lastCol) {
                    row += 1
                } else {
                    col += 1
                }
            } else {
                row -= 1
                col += 1
            }
        }
    }

    return result
}

fun isOutOfBounds(row: Int, col: Int, lastRow: Int, lastCol: Int): Boolean
{
    return row < 0 || row > lastRow || col < 0 || col > lastCol
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/zigzagTraverse/solution1/zigzagTraverse.kt

}

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/generateDivTags/solution1/generateDivTags.kt

```
package ae.hard.generateDivTags.solution1

fun generateDivTags(numberOfTags: Int): List<String> {
    val matchedDivTags = mutableListOf<String>()
    generateDivTagsHelper(numberOfTags, numberOfTags, listOf<String>(),
    matchedDivTags)
    return matchedDivTags
}

fun generateDivTagsHelper(opening: Int, closing: Int, prefix: List<String>,
    result: MutableList<String>) {
    if (opening > 0) {
        val newPrefix = prefix.plus("<div>")
        generateDivTagsHelper(opening - 1, closing, newPrefix, result)
    }

    // Means that more "opening" has been "used" than closing
    if (closing > opening) {
        val newPrefix = prefix.plus("</div>")
        generateDivTagsHelper(opening, closing - 1, newPrefix, result)
    }

    if (closing == 0) {
        result.add(prefix.joinToString(""))
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/knapsackProblem/solution1/knapsackProblem.kt

```
package ae.hard.knapsackProblem.solution1

import kotlin.math.max

fun knapsackProblem(items: List<List<Int>>, capacity: Int): Pair<Int, List<Int>> {
    val knapsack = mutableListOf(items.size + 1) { mutableListOf(capacity + 1) }
    for (i in 1 .. items.size) {
        val (currentValue, currentWeight) = items[i - 1]

        for (j in 0 .. capacity) {
            if (currentWeight > j) {
                knapsack[i][j] = knapsack[i - 1][j]
            } else {
                // knapsack[i - 1][j - currentWeight] + currentValue is
                // current value added with
                // The value when we didn't have current weight added
                knapsack[i][j] = max(knapsack[i - 1][j], knapsack[i - 1][j - currentWeight] + currentValue)
            }
        }
    }

    return Pair(knapsack[items.size][capacity], buildSequence(knapsack, items))
}

fun buildSequence(knapsack: MutableList<MutableList<Int>>, items: List<List<Int>>): List<Int> {
    val result = mutableListOf<Int>()
    var r = knapsack.lastIndex
    var c = knapsack[0].lastIndex

    while (r > 0) {
        if (knapsack[r][c] == knapsack[r - 1][c]) {
            r -= 1
        } else {
            result.add(r - 1)

            val weight = items[r - 1][1]
            c -= weight

            r -= 1
        }

        if (c == 0) {
            break
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/knapsackProblem/solution1/knapsackProblem.kt

```
    return result.reversed()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/shiftLinkedList/solution1/shiftLinkedList.kt

```
package ae.hard.shiftLinkedList.solution1

import kotlin.math.abs

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

// 0 -> 1 -> 2 -> 3 -> 4 -> 5, k = 2
// 4 -> 5 -> 0 -> 1 -> 2 -> 3
// 0 -> 1 -> 2 -> 3 -> 4 -> 5, k = -2
// 2 -> 3 -> 4 -> 5 -> 0 -> 1
fun shiftLinkedList(head: LinkedList, k: Int): LinkedList {
    var tail = head
    var listLength = 1

    while (tail.next != null) {
        tail = tail.next!!
        listLength += 1
    }

    val offset = abs(k) % listLength

    if (offset == 0) return head

    val newTailPosition = if (k > 0) listLength - offset else offset

    var newTail = head
    for (i in 1 until newTailPosition) {
        newTail = newTail.next!!
    }

    val newHead = newTail.next!!
    newTail.next = null
    tail.next = head

    return newHead
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/topologicalSort/solution1/topologicalSort.kt

```
package ae.hard.topologicalSort.solution1

fun topologicalSort(jobs: List<Int>, deps: List<List<Int>>): List<Int> {
    val jobGraph = createJobGraph(jobs, deps)
    return getSortedNodes(jobGraph)
}

fun createJobGraph(jobs: List<Int>, deps: List<List<Int>>): JobGraph {
    return JobGraph(jobs).also {
        deps.forEach { dep ->
            it.addDependent(dep[0], dep[1])
        }
    }
}

fun getSortedNodes(jobGraph: JobGraph): List<Int> {
    val sortedNodes = mutableListOf<Int>()

    val nodesWithNoPrereq = jobGraph.nodes.filter { it.prereqCount == 0 }.toMutableList()

    while (nodesWithNoPrereq.size > 0) {
        val node = nodesWithNoPrereq.removeAt(nodesWithNoPrereq.lastIndex)
        sortedNodes.add(node.job)
        removeDeps(node, nodesWithNoPrereq)
    }

    val stillContainEdges = jobGraph.nodes.any { it.prereqCount > 0 }
    if (stillContainEdges) return listOf()

    return sortedNodes
}

fun removeDeps(jobNode: JobNode, nodesWithNoPrereq: MutableList<JobNode>) {
    while (jobNode.deps.size > 0) {
        val dep = jobNode.deps.removeAt(jobNode.deps.lastIndex)
        dep.prereqCount -= 1
        if (dep.prereqCount == 0) {
            nodesWithNoPrereq.add(dep)
        }
    }
}

class JobGraph(val jobs: List<Int>) {
    val nodes = mutableListOf<JobNode>()
    val graph = mutableMapOf<Int, JobNode>()

    init {
        jobs.forEach {
            createNode(it)
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/topologicalSort/solution1/topologicalSort.kt

```
fun createNode(job: Int) {
    graph[job] = JobNode(job).also {
        nodes.add(it)
    }
}

fun addDependent(job: Int, dependent: Int) {
    val jobNode = getJobNode(job)
    val dependentNode = getJobNode(dependent)
    dependentNode.prereqCount += 1
    jobNode.deps.add(dependentNode)
}

fun getJobNode(job: Int): JobNode {
    if (!graph.containsKey(job)) {
        createNode(job)
    }
    return graph[job]!!
}

class JobNode(val job: Int) {
    val deps = mutableListOf<JobNode>()
    var prereqCount = 0
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/topologicalSort/solution2/topologicalSort.kt

```
package ae.hard.topologicalSort.solution2

fun topologicalSort(jobs: List<Int>, deps: List<List<Int>>): List<Int> {
    val jobGraph = buildJobGraph(jobs, deps)
    return getOrderedJobs(jobGraph)
}

fun buildJobGraph(jobs: List<Int>, deps: List<List<Int>>): JobGraph {
    return JobGraph(jobs).also { jobGraph ->
        deps.forEach { dep ->
            jobGraph.addPrereq(dep[1], dep[0])
        }
    }
}

fun getOrderedJobs(jobGraph: JobGraph): List<Int> {
    val orderedJobs = mutableListOf<Int>()
    val jobs = jobGraph.nodes

    while (jobs.isNotEmpty()) {
        val job = jobs.removeAt(jobs.lastIndex)
        val hasCycle = dfsJob(job, orderedJobs)
        if (hasCycle) return listOf()
    }
    return orderedJobs
}

fun dfsJob(jobNode: JobNode, orderedJobs: MutableList<Int>): Boolean {
    if (jobNode.visited) return false
    if (jobNode.visiting) return true

    jobNode.visiting = true
    jobNode.prereqs.forEach { prereq ->
        if (dfsJob(prereq, orderedJobs)) {
            return true
        }
    }

    jobNode.visited = true
    jobNode.visiting = false

    orderedJobs.add(jobNode.job)
    return false
}

class JobGraph(val jobs: List<Int>) {
    val nodes = mutableListOf<JobNode>()
    val graph = mutableMapOf<Int, JobNode>()

    init {
        jobs.forEach {
            addNode(it)
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/topologicalSort/solution2/topologicalSort.kt

```
    }

}

fun addNode(job: Int) {
    graph[job] = JobNode(job).also {
        nodes.add(it)
    }
}

fun addPrereq(job: Int, prereq: Int) {
    val jobNode = getNode(job)
    val prereqNode = getNode(prereq)
    jobNode.prereqs.add(prereqNode)
}

fun getNode(job: Int): JobNode {
    if (!graph.containsKey(job)) {
        addNode(job)
    }
    return graph[job]!!
}

class JobNode(val job: Int) {
    val prereqs = mutableListOf<JobNode>()
    var visited = false
    var visiting = false
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/continuousMedian/solution1/continuousMedian.kt

```
package ae.hard.continuousMedian.solution1

import java.util.PriorityQueue

open class ContinuousMedianHandler() {
    private var median: Double? = null

    val lowers = PriorityQueue<Int>(Comparator<Int> { i1, i2 ->
        i2.compareTo(i1)
    })
    val greater = PriorityQueue<Int>()

    fun insert(number: Int) {
        if (lowers.isEmpty() || number < lowers.peek()) {
            lowers.add(number)
        } else {
            greater.add(number)
        }

        rebalanceHeaps()
        updateMedian()
    }

    private fun rebalanceHeaps() {
        if (lowers.size - greater.size == 2) {
            greater.add(lowers.poll())
        } else if (greater.size - lowers.size == 2) {
            lowers.add(greater.poll())
        }
    }

    private fun updateMedian() {
        if (lowers.size == greater.size) {
            median = (lowers.peek() + greater.peek()) / 2.0
        } else if (lowers.size > greater.size) {
            median = lowers.peek().toDouble()
        } else {
            median = greater.peek().toDouble()
        }
    }

    fun getMedian(): Double? {
        return this.median
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/indexEqualsValue/solution1/indexEqualsValue.kt

```
package ae.hard.indexEqualsValue.solution1

fun indexEqualsValue(array: List<Int>): Int {
    return indexWithValueHelper(array, 0, array.lastIndex)
}

fun indexWithValueHelper(array: List<Int>, left: Int, right: Int): Int {
    if (left > right) return -1

    val mid = (left + right) / 2

    return if (array[mid] < mid) {
        indexWithValueHelper(array, mid + 1, right)
    } else if (array[mid] > mid) {
        indexWithValueHelper(array, left, mid - 1)
    } else {
        if (mid == 0 || array[mid - 1] < mid - 1) {
            mid
        } else {
            indexWithValueHelper(array, left, mid - 1)
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/indexEqualsValue/solution2/indexEqualsValue.kt

```
package ae.hard.indexEqualsValue.solution2

fun indexEqualsValue(array: List<Int>): Int {
    return indexWithValueHelper(array, 0, array.lastIndex)
}

fun indexWithValueHelper(array: List<Int>, left: Int, right: Int): Int {
    if (left > right) return -1

    val mid = (left + right) / 2

    return if (array[mid] < mid) {
        indexWithValueHelper(array, mid + 1, right)
    } else if (array[mid] == mid && mid == 0) {
        mid
    } else if (array[mid] == mid && array[mid - 1] < mid - 1) {
        mid
    } else {
        indexWithValueHelper(array, left, mid - 1)
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/indexEqualsValue/solution3/indexEqualsValue.kt

```
package ae.hard.indexEqualsValue.solution3

fun indexEqualsValue(array: List<Int>): Int {
    var left = 0
    var right = array.lastIndex

    while (left <= right) {
        val mid = (left + right) / 2

        if (array[mid] < mid) {
            left = mid + 1
        } else if (array[mid] == mid && mid == 0) {
            return mid
        } else if (array[mid] == mid && array[mid - 1] < mid - 1) {
            return mid
        } else {
            right = mid - 1
        }
    }

    return -1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/mergeLinkedLists/solution1/mergeLinkedLists.kt

```
package ae.hard.mergeLinkedLists.solution1

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun mergeLinkedLists(headOne: LinkedList, headTwo: LinkedList): LinkedList
{
    var p1: LinkedList? = headOne
    var p2: LinkedList? = headTwo
    var p1Prev: LinkedList? = null

    while (p1 != null && p2 != null) {
        if (p1.value < p2.value) {
            p1Prev = p1
            p1 = p1.next
        } else {
            if (p1Prev != null) {
                p1Prev.next = p2
            }

            p1Prev = p2
            p2 = p2.next
            p1Prev.next = p1
        }
    }

    if (p1 == null) {
        p1Prev?.next = p2
    }

    return if (headOne.value < headTwo.value) headOne else headTwo
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/mergeLinkedLists/solution2/mergeLinkedLists.kt

```
package ae.hard.mergeLinkedLists.solution2

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun mergeLinkedLists(headOne: LinkedList, headTwo: LinkedList): LinkedList
{
    recursiveMerge(headOne, headTwo, null)
    return if (headOne.value < headTwo.value) headOne else headTwo
}

fun recursiveMerge(p1: LinkedList?, p2: LinkedList?, p1Prev: LinkedList?) {
    if (p1 == null) {
        p1Prev?.next = p2
        return
    }

    if (p2 == null) {
        return
    }

    if (p1.value < p2.value) {
        recursiveMerge(p1.next, p2, p1)
    } else {
        if (p1Prev != null) {
            p1Prev.next = p2
        }

        val nextP2 = p2.next
        p2.next = p1
        recursiveMerge(p1, nextP2, p2)
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/minNumberOfJumps/solution1/minNumberOfJumps.kt

```
package ae.hard.minNumberOfJumps.solution1

import kotlin.math.min

fun minNumberOfJumps(array: List<Int>): Int {
    val jumps = MutableList(array.size) { Int.MAX_VALUE }
    jumps[0] = 0

    for (i in 0 until array.size) {
        for (j in 0 until i) {
            if (j + array[j] >= i) {
                jumps[i] = min(jumps[i], jumps[j] + 1)
            }
        }
    }

    return jumps.last()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/minNumberOfJumps/solution2/minNumberOfJumps.kt

```
package ae.hard.minNumberOfJumps.solution2

import kotlin.math.max

fun minNumberOfJumps(array: List<Int>): Int {

    if (array.size == 1) return 0

    var jumps = 0
    var maxReach = array[0]
    var steps = array[0]

    for (i in 1 until array.lastIndex) {
        maxReach = max(maxReach, i + array[i])
        steps -= 1

        if (steps == 0) {
            jumps += 1
            steps = maxReach - i
        }
    }

    return jumps + 1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/sortKSortedArray/solution1/sortKSortedArray.kt

```
package ae.hard.sortKSortedArray.solution1

import java.util.PriorityQueue
import kotlin.math.min

fun sortKSortedArray(array: MutableList<Int>, k: Int): MutableList<Int> {
    val minHeapWithKElements = PriorityQueue<Int>()

    // At each index we need to check k elements on it's right
    // Including itself, k + 1 elements as potential candidate
    // to be placed in this index.
    for (i in 0 until min(k + 1, array.size)) {
        minHeapWithKElements.add(array[i])
    }

    var nextIndexToInsertElement = 0

    for (i in k + 1 .. array.lastIndex) {
        // We get the min element from the heap of k + 1 size
        // And insert into the result array
        val minElement = minHeapWithKElements.poll()
        array[nextIndexToInsertElement] = minElement
        nextIndexToInsertElement += 1

        // And after that we push the current element
        // from the array into the heap to keep the heap
        // k + 1 elements long
        val currentElement = array[i]
        minHeapWithKElements.add(currentElement)
    }

    while (minHeapWithKElements.isNotEmpty()) {
        val minElement = minHeapWithKElements.poll()
        array[nextIndexToInsertElement] = minElement
        nextIndexToInsertElement += 1
    }

    return array
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/multiStringSearch/solution1/multiStringSearch.kt

```
package ae.hard.multiStringSearch.solution1

// O(bns) Time | O(n) Space
fun multiStringSearch(bigString: String, smallStrings: List<String>): List<Boolean> {
    return smallStrings.map { smallString ->
        isInBigString(bigString, smallString)
    }
}

fun isInBigString(bigString: String, smallString: String): Boolean {
    for (i in bigString.indices) {
        if (i + smallString.length > bigString.length) {
            break
        }

        if (isInBigStringHelper(bigString, smallString, i)) {
            return true
        }
    }

    return false
}

fun isInBigStringHelper(bigString: String, smallString: String, startIdx: Int): Boolean {
    var bigLeftIdx = startIdx
    var bigRightIdx = startIdx + smallString.length - 1

    var smallLeftIdx = 0
    var smallRightIdx = smallString.lastIndex

    while (bigLeftIdx < bigRightIdx) {
        if (bigString[bigLeftIdx] != smallString[smallLeftIdx] || bigString[bigRightIdx] != smallString[smallRightIdx]) {
            return false
        }

        bigLeftIdx += 1
        smallLeftIdx += 1
        bigRightIdx -= 1
        smallRightIdx -= 1
    }

    return true
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/multiStringSearch/solution2/multiStringSearch.kt

```
package ae.hard.multiStringSearch.solution2

// O(b^2 + ns) Time | O(b^2 + n) Space
fun multiStringSearch(bigString: String, smallStrings: List<String>): List<Boolean> {
    val suffixTrie = ModifiedSuffixTree(bigString)

    return smallStrings.map { smallString ->
        suffixTrie.contains(smallString)
    }
}

class TrieNode {
    val children = mutableMapOf<Char, TrieNode>()
}

class ModifiedSuffixTree(val str: String) {
    val rootNode = TrieNode()

    init {
        populateSuffixTrie(str)
    }

    fun populateSuffixTrie(str: String) {
        for (i in str.indices) {
            insertSubstringStartingAt(str, i)
        }
    }

    fun insertSubstringStartingAt(str: String, idx: Int) {
        var currentNode = rootNode

        for (j in idx .. str.lastIndex) {
            val letter = str[j]

            if (!currentNode.children.containsKey(letter)) {
                currentNode.children[letter] = TrieNode()
            }

            currentNode = currentNode.children[letter]!!
        }
    }

    fun contains(str: String): Boolean {
        var currentNode = rootNode

        for (i in str.indices) {
            val letter = str[i]

            if (!currentNode.children.containsKey(letter)) {
                return false
            }
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/multiStringSearch/solution2/multiStringSearch.kt

```
        currentNode = currentNode.children[letter]!!
    }

    return true
}
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/multiStringSearch/solution3/multiStringSearch.kt

```
package ae.hard.multiStringSearch.solution3

// O(ns + bs) Time | O(ns) Space
fun multiStringSearch(bigString: String, smallStrings: List<String>): List<Boolean> {
    val trie = Trie()

    smallStrings.forEach { smallString ->
        trie.insert(smallString)
    }

    val containsSet = mutableSetOf<String>()
    for (i in bigString.indices) {
        findSmallStringsIn(bigString, i, trie, containsSet)
    }

    return smallStrings.map { smallString ->
        containsSet.contains(smallString)
    }
}

class TrieNode {
    val children = mutableMapOf<Char, TrieNode>()
    var word: String? = null
}

class Trie {
    val rootNode = TrieNode()
    val endSymbol = '*'

    fun insert(str: String) {
        var currentNode = rootNode

        for (i in str.indices) {
            val letter = str[i]

            if (!currentNode.children.containsKey(letter)) {
                currentNode.children[letter] = TrieNode()
            }

            currentNode = currentNode.children[letter]!!
        }

        currentNode.children[endSymbol] = TrieNode().also {
            it.word = str
        }
    }
}

fun findSmallStringsIn(bigString: String, startIdx: Int, trie: Trie,
containsSet: MutableSet<String>) {
    var currentNode = trie.rootNode
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/multiStringSearch/solution3/multiStringSearch.kt

```
for (idx in startIdx .. bigString.lastIndex) {
    val letter = bigString[idx]

    if (!currentNode.children.containsKey(letter)) {
        break
    }
    currentNode = currentNode.children[letter]!!

    if (currentNode.children.containsKey(trie.endSymbol)) {
        containsSet.add(currentNode.children[trie.endSymbol]!!..word!!)
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/reverseLinkedList/solution1/reverseLinkedList.kt

```
package ae.hard.reverseLinkedList.solution1

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun reverseLinkedList(head: LinkedList): LinkedList {
    var prev: LinkedList? = null
    var current: LinkedList? = head

    while (current != null) {
        val next = current.next
        current.next = prev
        prev = current
        current = next
    }

    return prev!!
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/dijkstrasAlgorithm/solution1/dijkstrasAlgorithm.kt

```
package ae.hard.dijkstrasAlgorithm.solution1

import kotlin.math.min

fun dijkstrasAlgorithm(start: Int, edges: List<List<List<Int>>>): List<Int> {
    val numVertices = edges.size

    // Set all node's min distances to infinity (we don't know their
    // distances yet)
    val minDistances = MutableList(numVertices) { Integer.MAX_VALUE }

    // We know the starting node's min distance. It's 0.
    minDistances[start] = 0

    val visited = mutableSetOf<Int>()

    while (visited.size < numVertices) {

        // Get a node which is unvisited and has minimum distance
        val (minDistanceVertex, minDistance) =
            getUnvisitedMinDistanceVertex(minDistances, visited)

        // This node has infinity distance. All it's connected nodes will
        // have infinity distance too because they will pass through this node.
        if (minDistance == Integer.MAX_VALUE)
            break

        // Insert this node in visited set.
        visited.add(minDistanceVertex)

        // For all of it's connected vertices
        for ((destination, distanceToDestination) in edges[
minDistanceVertex]) {
            // This node was already visited before, so must had a shorter
            // path
            if (destination in visited) {
                continue
            }

            // Calculate minDistances to the connected node, current
            // minDistance + it's distance from current node.
            minDistances[destination] = min(minDistances[destination],
minDistance + distanceToDestination)
        }
    }

    return minDistances.map { distance ->
        if (distance == Integer.MAX_VALUE) -1 else distance
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/dijkstrasAlgorithm/solution1/dijkstrasAlgorithm.kt

```
fun getUnvisitedMinDistanceVertex(minDistances: List<Int>, visited: Set<Int>): Pair<Int, Int> {
    var minDistance = Integer.MAX_VALUE
    var minDistanceVertex = -1

    for (idx in 0 .. minDistances.lastIndex) {
        val distance = minDistances[idx]

        if (idx in visited) continue

        if (distance <= minDistance) {
            minDistanceVertex = idx
            minDistance = distance
        }
    }

    return minDistanceVertex to minDistance
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/dijkstrasAlgorithm/solution2/dijkstrasAlgorithm.kt

```
package ae.hard.dijkstrasAlgorithm.solution2

fun dijkstrasAlgorithm(start: Int, edges: List<List<List<Int>>>): List<Int>
> {
    val numVertices = edges.size

    // Initial min distance of all nodes to be infinity
    // From this array we will never remove items
    val minDistances = MutableList(numVertices) { Integer.MAX_VALUE }
    // Starting node will have a min distance of 0
    minDistances[start] = 0

    // Items from this heap will be removed once they are processed
    val minDistanceItems = (0 .. edges.lastIndex).map {
        Item(it, Int.MAX_VALUE)
    }.toMutableList()

    val minDistanceHeap = MinHeap(minDistanceItems)
    // Starting node will have a distance of 0
    minDistanceHeap.update(start, 0)

    // While we have more items in heap (means that more items in the heap)
    while (!minDistanceHeap.isEmpty()) {

        // Get the vertex of the item with minimum distance and distance to it
        val (minDistanceVertex, currentMinDistance) = minDistanceHeap.remove()!!

        // If the min distance is infinity then no luck. Next items will also be infinity
        if (currentMinDistance == Int.MAX_VALUE) {
            break
        }

        // For every neighbor of this vertex with min distance
        for ((destination, distanceToDestination) in edges[minDistanceVertex]) {

            // Calculate new distance as sum of current min distance and the distance from current node
            val newDistance = currentMinDistance + distanceToDestination

            // Current distance of this destination VERTEX is stored in minDistances
            val currentDistance = minDistances[destination]

            // But our calculated new distance can be better of course
            if (newDistance < currentDistance) {
                // If it is then update the minDistances array
                minDistances[destination] = newDistance
            }
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/dijkstrasAlgorithm/solution2/dijkstrasAlgorithm.kt

```
// And inside the heap update the min distance for the
destination VERTEX
        minDistanceHeap.update(destination, newDistance)
    }
}

return minDistances.map { distance ->
    if (distance == Integer.MAX_VALUE) -1 else distance
}
}

data class Item(val vertex: Int, val distance: Int)

open class MinHeap(val array: MutableList<Item>) {

    val heap: MutableList<Item>
    val vertexMap: MutableMap<Int, Int> = mutableMapOf()

    init {
        array.forEachIndexed { idx, item ->
            vertexMap[item.vertex] = idx
        }

        heap = buildHeap(array)
    }

    fun buildHeap(array: MutableList<Item>): MutableList<Item> {
        val firstParentIdx = (array.lastIndex - 1) / 2
        var parentIdx = firstParentIdx

        while (parentIdx >= 0) {
            siftDown(parentIdx, array.lastIndex, array)
            parentIdx -= 1
        }

        return array
    }

    fun siftDown(currentIdx: Int, endIdx: Int, heap: MutableList<Item>) {
        var mutableCurrentIdx = currentIdx

        var childOneIdx = mutableCurrentIdx * 2 + 1

        while (childOneIdx <= endIdx) {
            val childTwoIdx = if (mutableCurrentIdx * 2 + 2 <= endIdx)
                mutableCurrentIdx * 2 + 2 else -1
            val idxToSwap = if (childTwoIdx != -1 && heap[childTwoIdx].distance < heap[childOneIdx].distance) {
                childTwoIdx
            } else childOneIdx
```

```

        if (heap[idxToSwap].distance < heap[mutableCurrentIdx].distance)
    ) {
        swap(mutableCurrentIdx, idxToSwap, heap)
        mutableCurrentIdx = idxToSwap
        childOneIdx = mutableCurrentIdx * 2 + 1
    } else {
        // Already in the correct position
        return
    }
}

fun siftUp(currentIdx: Int) {
    var mutableCurrentIdx = currentIdx

    var parentIdx = (mutableCurrentIdx - 1) / 2

    while (mutableCurrentIdx > 0 && heap[mutableCurrentIdx].distance <
heap[parentIdx].distance) {
        swap(mutableCurrentIdx, parentIdx, heap)
        mutableCurrentIdx = parentIdx
        parentIdx = (mutableCurrentIdx - 1) / 2
    }
}

fun isEmpty() = heap.isEmpty()

fun remove(): Item? {
    swap(heap.lastIndex, 0, heap)
    val nodeToRemove = heap.removeAt(heap.lastIndex)

    val (vertex, _) = nodeToRemove
    // Let's remove the entry (location inside heap array) from
    vertexMap, because it's no more there
    vertexMap.remove(vertex)

    siftDown(0, heap.lastIndex, heap)
    return nodeToRemove
}

fun update(vertex: Int, value: Int) {
    // Which index in the heap array the item is located? Update that
item.
    heap[vertexMap[vertex]!!] = Item(vertex, value)
    siftUp(vertexMap[vertex]!!)
}

fun swap(i: Int, j: Int, heap: MutableList<Item>) {
    // After this swap
    // Vertex in item at i will be located in index j
    // Vertex in item at j will be located in index i
}

```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/dijkstrasAlgorithm/solution2/dijkstrasAlgorithm.kt

```
    vertexMap[heap[i].vertex] = j
    vertexMap[heap[j].vertex] = i
    val temp = heap[i]
    heap[i] = heap[j]
    heap[j] = temp
}
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/findNodesDistanceK/solution1/findNodesDistanceK.kt

```
package ae.hard.findNodesDistanceK.solution1

import java.util.LinkedList

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun findNodesDistanceK(tree: BinaryTree, target: Int, k: Int): List<Int> {
    val nodesToParents = mutableMapOf<Int, BinaryTree?>()
    populateNodesToParents(tree, nodesToParents, null)
    val targetNode = getNodeFromValue(nodesToParents, tree, target)
    return bfsForNodesDistanceK(targetNode, nodesToParents, k)
}

fun populateNodesToParents(node: BinaryTree?, nodesToParents: MutableMap<Int, BinaryTree?>, parent: BinaryTree?) {
    if (node != null) {
        nodesToParents[node.value] = parent
        populateNodesToParents(node.left, nodesToParents, node)
        populateNodesToParents(node.right, nodesToParents, node)
    }
}

fun getNodeFromValue(nodesToParents: Map<Int, BinaryTree?>, tree: BinaryTree, target: Int): BinaryTree {
    // Handles root case
    if (tree.value == target) return tree

    val parent = nodesToParents[target]!!

    if (parent.left != null && parent.left!!.value == target) {
        return parent.left!!
    }

    return parent.right!!
}

fun bfsForNodesDistanceK(root: BinaryTree, nodesToParents: MutableMap<Int, BinaryTree?>, k: Int): List<Int> {
    val queue = LinkedList<Pair<BinaryTree, Int>>()
    // Visited is needed for below
    // Suppose we added a parent of a node. So if visited is not present
    // Then we can add the child back in the queue.
    val visited = mutableSetOf<Int>()

    queue.add(root to 0)

    while (queue.isNotEmpty()) {
        val current = queue.poll()
```

```
val (currentNode, currentDistance) = current

visited.add(currentNode.value)

// When we find one element at distance k
// all remaining elements in the queue are at distance k.
if (currentDistance == k) {
    val result = mutableListOf<Int>(currentNode.value)
    while (queue.isNotEmpty()) {
        val (node, _) = queue.poll()
        result.add(node.value)
    }
    return result
}

val possibleDests = listOf(currentNode.left, currentNode.right,
nodesToParents[currentNode.value])
for (nextNode in possibleDests) {
    if (nextNode == null) continue
    if (visited.contains(nextNode.value)) continue
    queue.add(nextNode to currentDistance + 1)
}
return listOf()
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/findNodesDistanceK/solution2/findNodesDistanceK.kt

```
package ae.hard.findNodesDistanceK.solution2

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun findNodesDistanceK(tree: BinaryTree, target: Int, k: Int): List<Int> {
    val nodesAtDistanceK = mutableListOf<Int>()
    findDistanceFromNodeToTarget(tree, target, k, nodesAtDistanceK)
    return nodesAtDistanceK
}

fun findDistanceFromNodeToTarget(node: BinaryTree?, target: Int, k: Int,
nodesAtDistanceK: MutableList<Int>): Int {
    if (node == null) return -1

    if (node.value == target) {
        findNodesAtDistanceK(node, 0, k, nodesAtDistanceK)
        return 1
    }

    val leftDistance = findDistanceFromNodeToTarget(node.left, target, k,
nodesAtDistanceK)
    val rightDistance = findDistanceFromNodeToTarget(node.right, target, k,
, nodesAtDistanceK)

    if (leftDistance == k || rightDistance == k) {
        nodesAtDistanceK.add(node.value)
    }

    if (leftDistance != -1) {
        findNodesAtDistanceK(node.right, leftDistance + 1, k,
nodesAtDistanceK)
        return leftDistance + 1
    }

    if (rightDistance != -1) {
        findNodesAtDistanceK(node.left, rightDistance + 1, k,
nodesAtDistanceK)
        return rightDistance + 1
    }

    return -1
}

fun findNodesAtDistanceK(node: BinaryTree?, distance: Int, k: Int,
nodesAtDistanceK: MutableList<Int>) {
    if (node == null) return

    if (distance == k) {
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/findNodesDistanceK/solution2/findNodesDistanceK.kt

```
    nodesAtDistanceK.add(node.value)
} else {
    findNodesAtDistanceK(node.left, distance + 1, k, nodesAtDistanceK)
    findNodesAtDistanceK(node.right, distance + 1, k, nodesAtDistanceK)
}
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/maximizeExpression/solution1/maximizeExpression.kt

```
package ae.hard.maximizeExpression.solution1

import kotlin.math.max

fun maximizeExpression(array: List<Int>): Int {
    if (array.size < 4) return 0

    var maxValue = Int.MIN_VALUE

    for (a in 0 .. array.lastIndex) {
        val aValue = array[a]

        for (b in a + 1 .. array.lastIndex) {
            val bValue = array[b]

            for (c in b + 1 .. array.lastIndex) {
                val cValue = array[c]

                for (d in c + 1 .. array.lastIndex) {
                    val dValue = array[d]

                    maxValue = max(maxValue, evaluateExpress(aValue, bValue,
, cValue, dValue))
                }
            }
        }
    }

    return maxValue
}

fun evaluateExpress(a: Int, b: Int, c: Int, d: Int): Int {
    return a - b + c - d
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/maximizeExpression/solution2/maximizeExpression.kt

```
package ae.hard.maximizeExpression.solution2

import kotlin.math.max

fun maximizeExpression(array: List<Int>): Int {
    if (array.size < 4) return 0

    val maxOfA = mutableListOf(array.size) { 0 }
    maxOfA[0] = array[0]

    val maxOfAMinusB = mutableListOf(array.size) { 0 }
    maxOfAMinusB[0] = Int.MIN_VALUE

    val maxOfAMinusBPlusC = mutableListOf(array.size) { 0 }
    maxOfAMinusBPlusC[0] = Int.MIN_VALUE
    maxOfAMinusBPlusC[1] = Int.MIN_VALUE

    val maxOfAMinusBPlusCMinusD = mutableListOf(array.size) { 0 }
    maxOfAMinusBPlusCMinusD[0] = Int.MIN_VALUE
    maxOfAMinusBPlusCMinusD[1] = Int.MIN_VALUE
    maxOfAMinusBPlusCMinusD[2] = Int.MIN_VALUE

    for (a in 1 .. array.lastIndex) {
        val currentMax = max(maxOfA[a - 1], array[a])
        maxOfA[a] = currentMax
    }

    for (b in 1 .. array.lastIndex) {
        val currentMax = max(maxOfAMinusB[b - 1], maxOfA[b - 1] - array[b])
        maxOfAMinusB[b] = currentMax
    }

    for (c in 2 .. array.lastIndex) {
        val currentMax = max(maxOfAMinusBPlusC[c - 1], maxOfAMinusB[c - 1]
    ] + array[c])
        maxOfAMinusBPlusC[c] = currentMax
    }

    for (d in 3 .. array.lastIndex) {
        val currentMax = max(maxOfAMinusBPlusCMinusD[d - 1],
maxOfAMinusBPlusC[d - 1] - array[d])
        maxOfAMinusBPlusCMinusD[d] = currentMax
    }

    return maxOfAMinusBPlusCMinusD.last()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/validateThreeNodes/solution1/validateThreeNodes.kt

```
package ae.hard.validateThreeNodes.solution1

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null
}

fun validateThreeNodes(nodeOne: BST, nodeTwo: BST, nodeThree: BST): Boolean {
    if (isDescendent(nodeOne, nodeTwo)) {
        return isDescendent(nodeTwo, nodeThree)
    }

    if (isDescendent(nodeThree, nodeTwo)) {
        return isDescendent(nodeTwo, nodeOne)
    }

    return false
}

fun isDescendent(node: BST?, target: BST): Boolean {
    if (node == null) return false

    if (node.value == target.value) return true

    return if (target.value < node.value) {
        isDescendent(node.left, target)
    } else {
        isDescendent(node.right, target)
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/validateThreeNodes/solution2/validateThreeNodes.kt

```
package ae.hard.validateThreeNodes.solution2

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null
}

fun validateThreeNodes(nodeOne: BST, nodeTwo: BST, nodeThree: BST): Boolean {
    if (isDescendent(nodeOne, nodeTwo)) {
        return isDescendent(nodeTwo, nodeThree)
    }

    if (isDescendent(nodeThree, nodeTwo)) {
        return isDescendent(nodeTwo, nodeOne)
    }

    return false
}

fun isDescendent(node: BST?, target: BST): Boolean {
    var currentNode = node

    while (currentNode != null && currentNode != target) {
        currentNode = if (target.value < currentNode.value) {
            currentNode.left
        } else {
            currentNode.right
        }
    }

    return currentNode == target
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/validateThreeNodes/solution3/validateThreeNodes.kt

```
package ae.hard.validateThreeNodes.solution3

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null
}

fun validateThreeNodes(nodeOne: BST, nodeTwo: BST, nodeThree: BST): Boolean
{
    var searchOne: BST? = nodeOne
    var searchTwo: BST? = nodeThree

    while (true) {
        val findNodeThreeFromNodeOne = searchOne == nodeThree
        val findNodeOneFromNodeThree = searchTwo == nodeOne
        val foundNodeTwo = searchOne == nodeTwo || searchTwo == nodeTwo
        val finishedSearching = searchOne == null && searchTwo == null

        if (findNodeThreeFromNodeOne || findNodeOneFromNodeThree || foundNodeTwo || finishedSearching) {
            break
        }

        if (searchOne != null) {
            searchOne = if (nodeTwo.value < searchOne.value) searchOne.left
        } else searchOne.right
    }

        if (searchTwo != null) {
            searchTwo = if (nodeTwo.value < searchTwo.value) searchTwo.left
        } else searchTwo.right
    }

    val foundEachOther = searchOne == nodeThree || searchTwo == nodeOne
    val foundNodeTwo = searchOne == nodeTwo || searchTwo == nodeTwo

    if (!foundNodeTwo || foundEachOther) return false

    return searchForNode(
        nodeTwo,
        if (searchOne == nodeTwo) nodeThree else nodeOne
    )
}

fun searchForNode(node: BST, target: BST): Boolean {
    var currentNode: BST? = node

    while (currentNode != null && currentNode != target) {
        currentNode = if (target.value < currentNode.value) {
            currentNode.left
        }
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/validateThreeNodes/solution3/validateThreeNodes.kt

```
    } else {
        currentNode.right
    }
}

return currentNode == target
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/interweavingStrings/solution1/interweavingStrings.kt

```
package ae.hard.interweavingStrings.solution1

fun interweavingStrings(one: String, two: String, three: String): Boolean {
    if (one.length + two.length != three.length) {
        return false
    }
    return areInterwoven(one, two, three, 0, 0)
}

fun areInterwoven(one: String, two: String, three: String, i: Int, j: Int): Boolean {
    val k = i + j

    if (k == three.length) {
        return true
    }

    if (i < one.length && one[i] == three[k]) {
        if (areInterwoven(one, two, three, i + 1, j)) {
            return true
        }
    }

    if (j < two.length && two[j] == three[k]) {
        return areInterwoven(one, two, three, i, j + 1)
    }

    return false
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/interweavingStrings/solution2/interweavingStrings.kt

```
package ae.hard.interweavingStrings.solution2

fun interweavingStrings(one: String, two: String, three: String): Boolean {
    if (one.length + two.length != three.length) {
        return false
    }

    // Either one of i and j can go out of bound while the other is in
    // bound.
    // So we keep the cache size one more than the string sizes.
    val cache = MutableList(one.length + 1) {
        MutableList(two.length + 1) { -1 }
    }
    return areInterwoven(one, two, three, 0, 0, cache)
}

fun areInterwoven(one: String, two: String, three: String, i: Int, j: Int,
cache: MutableList<MutableList<Int>>): Boolean {
    if (cache[i][j] != -1) {
        return cache[i][j] == 1
    }

    val k = i + j

    if (k == three.length) {
        return true
    }

    if (i < one.length && one[i] == three[k]) {
        val areNextInterwoven = areInterwoven(one, two, three, i + 1, j,
cache)
        fillCachePosition(cache, i, j, areNextInterwoven)

        if (cache[i][j] == 1) {
            return true
        }
    }

    if (j < two.length && two[j] == three[k]) {
        val areNextInterwoven = areInterwoven(one, two, three, i, j + 1,
cache)
        fillCachePosition(cache, i, j, areNextInterwoven)
        return cache[i][j] == 1
    }

    cache[i][j] = 0
    return false
}

fun fillCachePosition(cache: MutableList<MutableList<Int>>, i: Int, j: Int
, value: Boolean) {
    cache[i][j] = if (value) 1 else 0
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/interweavingStrings/solution2/interweavingStrings.kt

}

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/lowestCommonManager/solution1/lowestCommonManager.kt

```
package ae.hard.lowestCommonManager.solution1

class OrgChart(name: Char) {
    val name = name
    val directReports = mutableListOf<OrgChart>()
}

fun getLowestCommonManager(topManager: OrgChart, reportOne: OrgChart,
reportTwo: OrgChart): OrgChart? {
    return getOrgInfo(topManager, reportOne, reportTwo).lowestCommonManager
}

data class OrgInfo(val lowestCommonManager: OrgChart?, val
numberOfImportantReports: Int)

fun getOrgInfo(manager: OrgChart, reportOne: OrgChart, reportTwo: OrgChart
): OrgInfo {
    var numberOfImportantReports = 0

    for (directReport in manager.directReports) {
        val reportOrgInfo = getOrgInfo(directReport, reportOne, reportTwo)
        if (reportOrgInfo.lowestCommonManager != null) {
            return reportOrgInfo
        }
        numberOfImportantReports += reportOrgInfo.numberOfImportantReports
    }

    if (manager == reportOne || manager == reportTwo) {
        numberOfImportantReports += 1
    }

    val lowestCommonManager = if (numberOfImportantReports == 2) manager
else null

    return OrgInfo(lowestCommonManager, numberOfImportantReports)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/maximumSumSubmatrix/solution1/maximumSumSubmatrix.kt

```
package ae.hard.maximumSumSubmatrix.solution1

import kotlin.math.max

fun maximumSumSubmatrix(matrix: List<List<Int>>, size: Int): Int {
    // Each element considered to be the bottom right corner of the
    rectangle
    // Formula sum = sums[r][c] - sums[r - size][c] - sums[r][c - size] +
    sums[r - size][c - size]
    val sumMatrix = createSumMatrix(matrix)
    var maxSum = Integer.MIN_VALUE

    // i = (size - 1) and j = (size - 1) constitutes the first
    // size * size rectangle whose bottom right corner is at (i, j)
    for (r in size - 1 until matrix.size) {
        for (c in size - 1 until matrix[0].size) {
            var total = sumMatrix[r][c]

            val touchesTopBorder = r - size < 0
            if (!touchesTopBorder) {
                total -= sumMatrix[r - size][c]
            }

            val touchesLeftBorder = c - size < 0
            if (!touchesLeftBorder) {
                total -= sumMatrix[r][c - size]
            }

            val touchesAnyBorder = touchesTopBorder || touchesLeftBorder
            if (!touchesAnyBorder) {
                total += sumMatrix[r - size][c - size]
            }

            maxSum = max(maxSum, total)
        }
    }

    return maxSum
}

fun createSumMatrix(matrix: List<List<Int>>): List<List<Int>> {
    val sumMatrix = mutableListOf(matrix.size) {
        mutableListOf(matrix[0].size) { 0 }
    }

    sumMatrix[0][0] = matrix[0][0]

    // Populate first row
    for (c in 1 until matrix[0].size) {
        sumMatrix[0][c] = matrix[0][c] + sumMatrix[0][c - 1]
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/maximumSumSubmatrix/solution1/maximumSumSubmatrix.kt

```
// Populate first column
for (r in 1 until matrix.size) {
    sumMatrix[r][0] = matrix[r][0] + sumMatrix[r - 1][0]
}

// sums[r][c] = matrix[r][c] + sumMatrix[r - 1][c] + sumMatrix[r][c - 1]
] - sumMatrix[r - 1][c - 1]
for (r in 1 until matrix.size) {
    for (c in 1 until matrix[0].size) {
        sumMatrix[r][c] = matrix[r][c] + sumMatrix[r - 1][c] +
sumMatrix[r][c - 1] - sumMatrix[r - 1][c - 1]
    }
}

return sumMatrix
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/shiftedBinarySearch/solution1/shiftedBinarySearch.kt

```
package ae.hard.shiftedBinarySearch.solution1

fun shiftedBinarySearch(array: List<Int>, target: Int): Int {
    return binarySearchHelper(array, target, 0, array.lastIndex)
}

fun binarySearchHelper(array: List<Int>, target: Int, left: Int, right:Int): Int {
    if (left > right) return -1

    val middle = (left + right) / 2
    val potentialMatch = array[middle]
    val leftNum = array[left]
    val rightNum = array[right]

    return if (target == potentialMatch) {
        middle
    } else if (leftNum <= potentialMatch) {
        // Left half of the array is sorted
        if (target in leftNum until potentialMatch) {
            binarySearchHelper(array, target, left, middle - 1)
        } else {
            binarySearchHelper(array, target, middle + 1, right)
        }
    } else {
        // Right half of the array is sorted
        if (target in (potentialMatch + 1)..rightNum) {
            binarySearchHelper(array, target, middle + 1, right)
        } else {
            binarySearchHelper(array, target, left, middle - 1)
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/shiftedBinarySearch/solution2/shiftedBinarySearch.kt

```
package ae.hard.shiftedBinarySearch.solution2

fun shiftedBinarySearch(array: List<Int>, target: Int): Int {
    var left = 0
    var right = array.lastIndex

    while (left <= right) {

        val middle = (left + right) / 2
        val potentialMatch = array[middle]
        val leftNum = array[left]
        val rightNum = array[right]

        if (target == potentialMatch) {
            return middle
        } else if (leftNum <= potentialMatch) {
            // Left half of the array is sorted
            if (target in leftNum until potentialMatch) {
                right = middle - 1
            } else {
                left = middle + 1
            }
        } else {
            // Right half of the array is sorted
            if (target in (potentialMatch + 1)..rightNum) {
                left = middle + 1
            } else {
                right = middle - 1
            }
        }
    }

    return -1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/ambiguousMeasurements/solution1/ambiguousMeasurements.kt

```
package ae.hard.ambiguousMeasurements.solution1

import kotlin.math.max

fun ambiguousMeasurements(measuringCups: List<List<Int>>, low: Int, high: Int): Boolean {
    // Let range we need to measure is [2200, 2000]
    // Let us think we have two cups [[200, 300], [400, 500]]
    //
    //           2200, 2000
    //           /   \
    //           /   \
    // Considering [200, 300] Considering [400, 500]
    //           /       \
    //           [2000, 1700] [1800, 1500]
    //           /           \
    //           /           \
    // Considering [400, 500] considering [200, 300]
    //           /           \
    //           [1600, 1200]     [1600, 1200]
    // From above we see that we arrived at [1600, 1200] from both of the
    branches.
    // To avoid such issue we have to implement memoization.
    return canMeasureInRange(measuringCups, low, high, mutableMapOf())
}

fun canMeasureInRange(measuringCups: List<List<Int>>, low: Int, high: Int,
memo: MutableMap<String, Boolean>): Boolean {
    val key = createHashableKey(low, high)

    // If we found the result with current low and high in memo, then
    return it
    if (memo.containsKey(key)) return memo[key]!!

    // Initialize can measure with false
    var canMeasure = false

    // We will keep recursing until both low and high reduces to 0, which
    is the base case.
    if (low == 0 && high == 0) return false

    // Then try with each measuring cup and try to measure with it
    for ((measuringCupLow, measuringCupHigh) in measuringCups) {
        // We can measure if measuring cup lies within the range defined by
        low and high
        // As example [100, 200] cup can measure [50, 250]
        //           [100, 200] cup can measure [100, 200]
        //           [100, 200] cup can't measure [1800, 2000]
        //           [100, 200] cup can't measure [150, 250]
        //           [100, 200] cup can't measure [150, 180]
        if (low <= measuringCupLow && high >= measuringCupHigh) {
            canMeasure = true
            break
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/ambiguousMeasurements/solution1/ambiguousMeasurements.kt

```
}

// newLow is current low minus the current cup's low mark
// newHigh is current high minus the current cup's high mark
// both newLow and newHigh should be bounded by 0
// Because practically there are no measurements less than 0.
// So if we have for example [-10, 10], we can make it [0, 10].
// Similarly if we have [100, -10] we can cap it to [100, 0].
// Because they are the same thing as definitely they will be less
// than any cup's low mark.
val newLow = max(low - measuringCupLow, 0)
val newHigh = max(high - measuringCupHigh, 0)

// Call canMeasureInRange recursively with newLow, newHigh
canMeasure = canMeasureInRange(measuringCups, newLow, newHigh, memo
)
    // If any of the recursive calls return true that means we can
    measure the volume. No need to do further computation. Store it in memo and
    pop true to the above layers.
    if (canMeasure) break
}

// Store the result in memo.
memo[key] = canMeasure

return memo[key]!!
}

fun createHashableKey(low: Int, high: Int): String {
    return "$low-$high"
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/underscorifySubstring/solution1/underscorifySubstring.kt

```
package ae.hard.underscorifySubstring.solution1

fun underscorifySubstring(string: String, substring: String): String {
    val mergedLocations = mergeLocations(getLocations(string, substring))
    return underscorify(string, mergedLocations)
}

// Merge intervals algorithm (for locations)
fun mergeLocations(locations: List<List<Int>>): List<List<Int>> {
    // Edge case - if locations array is empty then return
    if (locations.isEmpty()) return locations

    // Push first interval into merged intervals
    val mergedLocations = MutableList<List<Int>>(1) { locations.first() }

    // For all other locations
    for (i in 1 until locations.size) {
        // Get current location
        val currentLocation = locations[i]
        val lastLocation = mergedLocations.last()

        // If the locations does not overlap then push the current location
        // into merged location.
        if (currentLocation[0] > lastLocation[1]) {
            mergedLocations.add(currentLocation)
            // If they overlap then change the end location of the last
            // merged location.
        } else {
            mergedLocations[mergedLocations.lastIndex] = listOf(
                lastLocation[0], currentLocation[1])
        }
    }

    return mergedLocations
}

// Get locations of substr inside str using language default find function.
fun getLocations(string: String, substring: String): List<List<Int>> {
    val locations = mutableListOf<List<Int>>()

    // Start from index 0.
    var startIdx = 0

    // While we have more valid start index position
    while (startIdx < string.length) {

        // Find the position of substr inside of str starting from start
        // index using language default find function
        val nextIdx = string.indexOf(substring, startIdx)

        // If substring can't be found inside str starting from start index
        // , then it can't be found anymore, so break.
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/underscorifySubstring/solution1/underscorifySubstring.kt

```
if (nextIdx == -1) break

    // Otherwise insert { nextIdx, nextIdx + substring size } inside of
locations.
    locations.add(listOf(nextIdx, nextIdx + substring.length))

    // Then set startIdx is the next index of the first index of the
substr location.
    // abcdddeeffabcff, substr = abc
    // For first instance of abc, next startIdx will be 1 (b)
    startIdx = nextIdx + 1
}

return locations
}

fun underscorify(string: String, locations: List<List<Int>>): String {
    // Pointer inside string
    var strIdx = 0

    // Pointer inside locations array
    var locationIdx = 0

    // Underscorified result string
    val finalChars = mutableListOf<Char>()

    // If current character is between underscores. Initially it is false.
    var inBetweenUnderscores = false

    // The index inside the interval under consideration.
    // Either 0 or 1, means either starting index or ending index.
    // Initially starting index.
    var i = 0

    // As long we are inside of the string and inside of the locations
array
    while (strIdx < string.length && locationIdx < locations.size) {
        val currentLocation = locations[locationIdx]

        // If current string index is equal to current interval's current
index (0 or 1)
        // Either start or end index
        if (strIdx == currentLocation[i]) {
            // Insert _ inside result.
            finalChars.add('_')

            // Revert inBetweenUnderscores.
            // If starting an interval then revert inBetweenUnderscores to
true.
            // If ending an interval then revert inBetweenUnderscores to
false.
            inBetweenUnderscores = !inBetweenUnderscores
        }
    }
}
```

```

        // Revert the index inside of the interval.
        // If starting an interval then toggle it to 1, otherwise
        toggle it to 0.
        i = if (i == 0) 1 else 0

        // If inBetweenUnderscores is reverted back to false, then it
        means that we are done with previous interval
        // Now we will consider the next interval. So we will increment
        the index of the location.
        if (!inBetweenUnderscores) {
            locationIdx += 1
        }
    }

    // Regardless we will insert current character into result array
    and increment the index to the next character.
    finalChars.add(string[strIdx])
    strIdx += 1
}

// Consider this case
// str = test abctest, substr = test
// Result of this should be _test_ abc_test_
// Now from string the locations returned will be like
// [0, 4], [8, 12]
// Now from the [8, 12] interval, we will break from loop before we
reach 12 (because it is str.size())
// So the inner logic will never run to insert the last underscore.
// And here we will be at locations.size() - 1.
// So we need to insert the last underscore in this case.
if (locationIdx < locations.size) {
    finalChars.add('_')
}

// Consider this case,
// str = test abc, substr = test
// Result of this should be _test_ abc
// After the occurrence of substr we will break out from the loop.
// So to insert the remaining " abc", we need to run the below loop and
and them to result.
while (strIdx < string.length) {
    finalChars.add(string[strIdx])
    strIdx += 1
}

// Construct a string from result vector and return.
return String(finalChars.toCharArray())
}

```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/maxPathSumInBinaryTree/solution1/maxPathSumInBinaryTree.kt

```
package ae.hard.maxPathSumInBinaryTree.solution1

import kotlin.math.max

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun maxPathSum(tree: BinaryTree): Int {
    return findMaxSum(tree).second
}

fun findMaxSum(node: BinaryTree?): Pair<Int, Int> {
    node ?: return 0 to Int.MIN_VALUE

    val (leftSubtreeBranchMaxSum, leftSubtreeMaxSum) = findMaxSum(node.left)
    val (rightSubtreeBranchMaxSum, rightSubtreeMaxSum) = findMaxSum(node.right)

    val value = node.value
    val childBranchMaxSum = max(leftSubtreeBranchMaxSum,
        rightSubtreeBranchMaxSum)
    val currentBranchMaxSum = max(childBranchMaxSum + value, value)
    val currentMaxSumAsRoot = max(leftSubtreeBranchMaxSum + value +
        rightSubtreeBranchMaxSum, currentBranchMaxSum)
    val currentMaxSum = max(
        max(leftSubtreeMaxSum, rightSubtreeMaxSum),
        currentMaxSumAsRoot
    )
    return currentBranchMaxSum to currentMaxSum
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/longestCommonSubsequence/solution1/longestCommonSubsequence.kt

```
package ae.hard.longestCommonSubsequence.solution1

// O(NM * min(N, M))
fun longestCommonSubsequence(str1: String, str2: String): List<Char> {
    val dp = MutableList(str1.length + 1) {
        MutableList(str2.length + 1) { listOf<Char>() }
    }

    for (i in 1 .. str1.length) {
        for (j in 1 .. str2.length) {
            if (str1[i - 1] != str2[j - 1]) {
                dp[i][j] = if (dp[i - 1][j].size > dp[i][j - 1].size) {
                    dp[i - 1][j].toList()
                } else {
                    dp[i][j - 1].toList()
                }
            } else {
                dp[i][j] = dp[i - 1][j - 1].plus(str1[i - 1])
            }
        }
    }

    return dp[str1.length][str2.length]
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/maxSumIncreasingSubsequence/solution1/maxSumIncreasingSubsequence.kt

```
package ae.hard.maxSumIncreasingSubsequence.solution1

fun maxSumIncreasingSubsequence(array: List<Int>): Pair<Int, List<Int>> {
    val sequences = MutableList(array.size) { -1 }
    val sums = array.toMutableList()

    var maxSumIdx = 0

    for (i in 0 .. array.lastIndex) {

        val currentNum = array[i]

        for (j in 0 until i) {
            val otherNum = array[j]

            // If we add currentNum with whatever sum is available at index
            j
            // and if it is more what whatever sum is available at index i
            // Then at index i store sum available at index j plus
            currentNum
            if (currentNum > otherNum && sums[j] + currentNum >= sums[i]) {
                sums[i] = sums[j] + currentNum
                sequences[i] = j
            }
        }

        if (sums[i] > sums[maxSumIdx]) {
            maxSumIdx = i
        }
    }

    return Pair(sums[maxSumIdx], buildSequences(array, sequences, maxSumIdx))
}

fun buildSequences(array: List<Int>, sequences: List<Int>, currentIdx: Int): List<Int> {
    val sequence = mutableListOf<Int>()
    var currentIdxMutable = currentIdx

    while (currentIdxMutable != -1) {
        sequence.add(array[currentIdxMutable])
        currentIdxMutable = sequences[currentIdxMutable]
    }

    sequence.reverse()

    return sequence
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/largestRectangleUnderSkyline/solution1/largestRectangleUnderSkyline.kt

```
package ae.hard.largestRectangleUnderSkyline.solution1

import kotlin.math.max

fun largestRectangleUnderSkyline(buildings: List<Int>): Int {
    var maxArea = 0

    for (i in buildings.indices) {
        val currentPillarHeight = buildings[i]

        var furthestLeftIdx = i
        while (furthestLeftIdx > 0 && buildings[furthestLeftIdx - 1] >= currentPillarHeight) {
            furthestLeftIdx -= 1
        }

        var furthestRightIdx = i
        while (furthestRightIdx < buildings.lastIndex && buildings[furthestRightIdx + 1] >= currentPillarHeight) {
            furthestRightIdx += 1
        }

        val currentArea = (furthestRightIdx - furthestLeftIdx + 1) * currentPillarHeight
        maxArea = max(maxArea, currentArea)
    }

    return maxArea
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/largestRectangleUnderSkyline/solution2/largestRectangleUnderSkyline.kt

```
package ae.hard.largestRectangleUnderSkyline.solution2

import kotlin.math.max

fun largestRectangleUnderSkyline(buildings: List<Int>): Int {
    var maxArea = 0
    val pillarIdxs = mutableListOf<Int>()

    // To clear out stack at the end of the algorithm
    val extendedBuildings = buildings.plus(0)

    for (i in extendedBuildings.indices) {
        val height = extendedBuildings[i]

        // Remove elements from top of the stack as long as
        // Top element from stack has a greater than or equal to the height
        // of the current element pointed with i
        while (pillarIdxs.isNotEmpty() && extendedBuildings[pillarIdxs.last()] >= height) {
            val pillarHeight = extendedBuildings[pillarIdxs.removeAt(
                pillarIdxs.lastIndex)]

            // i is the right bound
            // pillarIdxs.back() is the left bound
            // Lower element on stack always should have smaller value
            // If not then they must have been removed already
            // both right and left bounds are indices which we can't
        include
            // So we do rightBound - leftBound - 1
            val width = if (pillarIdxs.isEmpty()) {
                i
            } else {
                i - pillarIdxs.last() - 1
            }

            maxArea = max(maxArea, pillarHeight * width)
        }

        pillarIdxs.add(i)
    }

    return maxArea
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/longestSubstringWithoutDuplication/solution1/longestSubstringWithoutDuplication

```
package ae.hard.longestSubstringWithoutDuplication.solution1

fun longestSubstringWithoutDuplication(string: String): String {
    var left = 0
    var right = 0

    var longestSubstr = 0 to 1

    val freq = mutableMapOf<Char, Int>()

    while (right < string.length) {
        val rightChar = string[right]

        if (!freq.containsKey(rightChar)) {
            freq[rightChar] = 0
        }
        freq[rightChar] = freq[rightChar]!! + 1

        while (freq[rightChar]!! > 1) {
            val leftChar = string[left]
            freq[leftChar] = freq[leftChar]!! - 1
            left += 1
        }

        if (right + 1 - left > longestSubstr.second - longestSubstr.first)
        {
            longestSubstr = left to right + 1
        }

        right += 1
    }

    return string.substring(longestSubstr.first, longestSubstr.second)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/hard/longestSubstringWithoutDuplication/solution2/longestSubstringWithoutDuplication

```
package ae.hard.longestSubstringWithoutDuplication.solution2

import kotlin.math.max

fun longestSubstringWithoutDuplication(string: String): String {
    val lastSeen = mutableMapOf<Char, Int>()
    var longest = 0 to 1
    var startIdx = 0

    for (i in string.indices) {
        val char = string[i]
        if (char in lastSeen) {
            startIdx = max(startIdx, lastSeen[char]!! + 1)
        }

        if (i + 1 - startIdx > longest.second - longest.first) {
            longest = startIdx to i + 1
        }
        lastSeen[char] = i
    }

    return string.substring(longest.first, longest.second)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/powerset/solution1/powerset.kt

```
package ae.medium.powerset.solution1

fun powerset(array: List<Int>): List<List<Int>> {
    val subsets = mutableListOf<List<Int>>(listOf())

    for (num in array) {
        val size = subsets.size

        for (i in 0 until size) {
            val currentSubset = subsets[i]
            val newSubset = currentSubset.plus(num)
            subsets.add(newSubset)
        }
    }

    return subsets
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/powerset/solution2/powerset.kt

```
package ae.medium.powerset.solution2

fun powerset(array: List<Int>): List<List<Int>> {
    return powersetHelper(array, 0)
}

fun powersetHelper(array: List<Int>, idx: Int): MutableList<List<Int>> {
    if (idx >= array.size) return mutableListOf(listOf())

    val item = array[idx]
    val subsets = powersetHelper(array, idx + 1)

    val size = subsets.size

    for (i in 0 until size) {
        val currentSubset = subsets[i]
        subsets.add(currentSubset.plus(item))
    }
}

return subsets
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/sortStack/solution1/sortStack.kt

```
package ae.medium.sortStack.solution1

fun sortStack(stack: MutableList<Int>): MutableList<Int> {
    if (stack.isEmpty()) {
        return stack
    }

    val top = stack.last()
    stack.removeAt(stack.lastIndex)

    sortStack(stack)
    insertInSortedStack(stack, top)

    return stack
}

fun insertInSortedStack(stack: MutableList<Int>, value: Int) {
    if (stack.isEmpty() || value >= stack.last()) {
        stack.add(value)
        return
    }

    val top = stack.last()
    stack.removeAt(stack.lastIndex)

    insertInSortedStack(stack, value)

    stack.add(top)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/riverSizes/solution1/riverSizes.kt

```
package ae.medium.riverSizes.solution1

fun riverSizes(matrix: List<List<Int>>): List<Int> {
    val riverSizes = mutableListOf<Int>()

    val rows = matrix.size
    val cols = matrix[0].size
    val visited = MutableList(rows) { MutableList(cols) { false } }

    for (r in 0 until rows) {
        for (c in 0 until cols) {
            if (visited[r][c]) {
                continue
            }

            visitNode(r, c, matrix, visited, riverSizes)
        }
    }

    return riverSizes
}

fun visitNode(r: Int, c: Int, graph: List<List<Int>>, visited: MutableList<MutableList<Boolean>>, riverSizes: MutableList<Int>) {
    var currentRiverSize = 0

    val stack = mutableListOf<Pair<Int, Int>>()
    stack.add(r to c)

    while (stack.isNotEmpty()) {
        val (currentRow, currentCol) = stack.last()
        stack.removeAt(stack.lastIndex)

        if (visited[currentRow][currentCol]) continue

        if (graph[currentRow][currentCol] == 0) continue

        visited[currentRow][currentCol] = true
        currentRiverSize += 1

        for (neighbor in getUnvisitedNeighbors(currentRow, currentCol, visited)) {
            stack.add(neighbor)
        }
    }

    if (currentRiverSize > 0) {
        riverSizes.add(currentRiverSize)
    }
}

fun getUnvisitedNeighbors(r: Int, c: Int, visited: MutableList<MutableList<
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/riverSizes/solution1/riverSizes.kt

```
Boolean>>): List<Pair<Int, Int>> {
    return listOf(r - 1 to c, r to c - 1, r + 1 to c, r to c + 1).filter
{ (row, col) ->
    row >= 0 && col >= 0 && row < visited.size && col < visited[0].size
&& !visited[row][col]
}
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/longestPeak/solution1/longestPeak.kt

```
package ae.medium.longestPeak.solution1

import kotlin.math.max

fun longestPeak(array: List<Int>): Int {

    var longestPeakWidth = 0

    var i = 1

        // Loop starts from 1 and goes till array.size() - 1, so every element
        has both i - 1 and i + 1 positions
        while (i < array.lastIndex) {

            // If an element is greater than both of it's previous and next
            elements then we have a peak
            val isPeak = array[i - 1] < array[i] && array[i] > array[i + 1]

            // If current element is not a peak then continue
            if (!isPeak) {
                i += 1
                continue
            }

            // LeftIdx starts from 2 elements before the peak
            var leftIdx = i - 2
            // When leftIdx is greater than zero and it continues the peak
            // i.e it's smaller than it's next element, till then decrement it
            // i.e try to grow the peak to the left
            while (leftIdx >= 0 && array[leftIdx] < array[leftIdx + 1]) {
                leftIdx -= 1
            }

            // RightIdx starts from 2 elements after the peak
            var rightIdx = i + 2
            // When rightIdx is less than array size and it continues the peak
            // i.e it's smaller than it's previous element, till then increment
            it
            // i.e try to grow the peak to the right
            while (rightIdx < array.size && array[rightIdx] < array[rightIdx -
            1]) {
                rightIdx += 1
            }

            // Peak width calculation formula
            // rightIdx and leftIdx at this point isn't contained in the peak
            // If they were contained in the peak then the formula could be
            rightIdx - leftIdx + 1.
            val currentPeakWidth = rightIdx - leftIdx - 1
            longestPeakWidth = max(currentPeakWidth, longestPeakWidth)

            // Once the current peak is over, rightIdx will be the potential
```

```
File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/longestPeak/solution1/longestPeak.kt
next peak
    // Try to grow it and potentially update longest peak width in next
iteration
    i = rightIdx
}

return longestPeakWidth
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/sunsetViews/solution1/sunsetViews.kt

```
package ae.medium.sunsetViews.solution1

import kotlin.math.max

fun sunsetViews(buildings: List<Int>, direction: String): List<Int> {
    val range = if (direction == "EAST") {
        buildings.lastIndex downTo 0
    } else {
        0 .. buildings.lastIndex
    }

    var runningMax = 0
    val buildingsWithSunsetView = mutableListOf<Int>()

    for (idx in range) {
        val currentHeight = buildings[idx]

        if (currentHeight > runningMax) {
            buildingsWithSunsetView.add(idx)
        }

        runningMax = max(runningMax, currentHeight)
    }

    if (direction == "EAST") {
        buildingsWithSunsetView.reverse()
    }

    return buildingsWithSunsetView
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/sunsetViews/solution2/sunsetViews.kt

```
package ae.medium.sunsetViews.solution2

fun sunsetViews(buildings: List<Int>, direction: String): List<Int> {

    val range = if (direction == "EAST") {
        0 .. buildings.lastIndex
    } else {
        buildings.lastIndex downTo 0
    }

    val buildingsWithSunsetView = mutableListOf<Int>()

    for (idx in range) {
        val currentHeight = buildings[idx]

        // The items which current element can block we should pop them off
        while (buildingsWithSunsetView.isNotEmpty() && currentHeight >=
            buildings[buildingsWithSunsetView.last()]) {
            buildingsWithSunsetView.removeAt(buildingsWithSunsetView.
lastIndex)
        }

        buildingsWithSunsetView.add(idx)
    }

    if (direction == "WEST") {
        buildingsWithSunsetView.reverse()
    }

    return buildingsWithSunsetView
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/validateBST/solution1/validateBST.kt

```
package ae.medium.validateBST.solution1

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null
}

fun validateBst(tree: BST): Boolean {
    return validate(tree, null, null)
}

fun validate(node: BST?, max: BST?, min: BST?): Boolean {
    node ?: return true

    if (max != null && node.value >= max.value) return false
    if (min != null && node.value < min.value) return false

    return validate(node.left, node, min) && validate(node.right, max, node)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/validateBST/solution2/validateBST.kt

```
package ae.medium.validateBST.solution2

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null
}

fun validateBst(tree: BST): Boolean {
    return validate(tree, Integer.MAX_VALUE, Integer.MIN_VALUE)
}

fun validate(node: BST?, maxValue: Int, minValue: Int): Boolean {
    node ?: return true

    if (node.value >= maxValue || node.value < minValue) return false

    return validate(node.left, node.value, minValue) && validate(node.right,
        maxValue, node.value)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/bstTraversal/solution1/bstTraversal.kt

```
package ae.medium.bstTraversal.solution1

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null
}

fun inOrderTraverse(tree: BST?, array: MutableList<Int>): List<Int> {
    tree ?: return array

    inOrderTraverse(tree.left, array)
    array.add(tree.value)
    inOrderTraverse(tree.right, array)
    return array
}

fun preOrderTraverse(tree: BST?, array: MutableList<Int>): List<Int> {
    tree ?: return array

    array.add(tree.value)
    preOrderTraverse(tree.left, array)
    preOrderTraverse(tree.right, array)
    return array
}

fun postOrderTraverse(tree: BST?, array: MutableList<Int>): List<Int> {
    tree ?: return array

    postOrderTraverse(tree.left, array)
    postOrderTraverse(tree.right, array)
    array.add(tree.value)
    return array
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/cycleInGraph/solution1/cycleInGraph.kt

```
package ae.medium.cycleInGraph.solution1

fun cycleInGraph(edges: List<List<Int>>): Boolean {
    val numNodes = edges.size

    val visited = MutableList(numNodes) { false }
    val currentlyInStack = MutableList(numNodes) { false }

    for (node in 0 until numNodes) {
        if (visited[node]) {
            continue
        }

        if (isNodeInCycle(node, edges, visited, currentlyInStack)) {
            return true
        }
    }

    return false
}

fun isNodeInCycle(node: Int, edges: List<List<Int>>, visited: MutableList<Boolean>, currentlyInStack: MutableList<Boolean>): Boolean {
    visited[node] = true
    currentlyInStack[node] = true

    for (neighbor in edges[node]) {
        if (!visited[neighbor]) {
            if (isNodeInCycle(neighbor, edges, visited, currentlyInStack
)) {
                return true
            }
        } else if (currentlyInStack[neighbor]) {
            return true
        }
    }

    currentlyInStack[node] = false
    return false
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minHeightBST/solution1/minHeightBST.kt

```
package ae.medium.minHeightBST.solution1

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null

    fun insert(value: Int) {
        if (value < this.value) {
            if (this.left == null) {
                this.left = BST(value)
            } else {
                this.left!!.insert(value)
            }
        } else {
            if (this.right == null) {
                this.right = BST(value)
            } else {
                this.right!!.insert(value)
            }
        }
    }

    fun minHeightBst(array: List<Int>): BST {
        return constructMinHeightBST(array, null, 0, array.lastIndex)!!
    }

    fun constructMinHeightBST(array: List<Int>, bst: BST?, startIdx: Int,
endIdx: Int): BST? {
        if (endIdx < startIdx) return null

        var rootBst = bst
        val midIdx = (startIdx + endIdx) / 2
        val valueToAdd = array[midIdx]

        if (rootBst == null) {
            rootBst = BST(valueToAdd)
        } else {
            rootBst.insert(valueToAdd)
        }

        constructMinHeightBST(array, rootBst, startIdx, midIdx - 1)
        constructMinHeightBST(array, rootBst, midIdx + 1, endIdx)

        return rootBst
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minHeightBST/solution2/minHeightBST.kt

```
package ae.medium.minHeightBST.solution2

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null

    fun insert(value: Int) {
        if (value < this.value) {
            if (this.left == null) {
                this.left = BST(value)
            } else {
                this.left!!.insert(value)
            }
        } else {
            if (this.right == null) {
                this.right = BST(value)
            } else {
                this.right!!.insert(value)
            }
        }
    }

    fun minHeightBst(array: List<Int>): BST {
        return constructMinHeightBST(array, null, 0, array.lastIndex)!!
    }

    fun constructMinHeightBST(array: List<Int>, bst: BST?, startIdx: Int,
endIdx: Int): BST? {
        if (endIdx < startIdx) return null

        val midIdx = (startIdx + endIdx) / 2
        val newBSTNode = BST(array[midIdx])

        if (bst != null) {
            if (array[midIdx] < bst.value) {
                bst.left = newBSTNode
            } else {
                bst.right = newBSTNode
            }
        }

        constructMinHeightBST(array, newBSTNode, startIdx, midIdx - 1)
        constructMinHeightBST(array, newBSTNode, midIdx + 1, endIdx)

        return newBSTNode
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minHeightBST/solution3/minHeightBST.kt

```
package ae.medium.minHeightBST.solution3

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null

    fun insert(value: Int) {
        if (value < this.value) {
            if (this.left == null) {
                this.left = BST(value)
            } else {
                this.left!!.insert(value)
            }
        } else {
            if (this.right == null) {
                this.right = BST(value)
            } else {
                this.right!!.insert(value)
            }
        }
    }
}

fun minHeightBst(array: List<Int>): BST {
    return constructMinHeightBST(array, 0, array.lastIndex)!!
}

fun constructMinHeightBST(array: List<Int>, startIdx: Int, endIdx: Int): BST? {
    if (endIdx < startIdx) return null

    val midIdx = (startIdx + endIdx) / 2
    val newBSTNode = BST(array[midIdx])

    newBSTNode.left = constructMinHeightBST(array, startIdx, midIdx - 1)
    newBSTNode.right = constructMinHeightBST(array, midIdx + 1, endIdx)

    return newBSTNode
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/permutations/solution1/permutations.kt

```
package ae.medium.permutations.solution1

// [1, 2, 3]
// permutationsHelper(0, [1, 2, 3], permutations)
//   j = 0
//   [1, 2, 3]
//   permutationsHelper(1, [1, 2, 3], permutations)
//     j = 1
//     [1, 2, 3]
//     permutationsHelper(2, [1, 2, 3], permutations)
//       j = 2
//       [1, 2, 3]
//       permutationsHelper(3, [1, 2, 3], permutations) - PUSH
//       [1, 2, 3]
//       [1, 2, 3]
//
//       j = 2
//       [1, 3, 2]
//       permutationsHelper(2, [1, 3, 2], permutations)
//         j = 2
//         [1, 3, 2]
//         permutationsHelper(3, [1, 3, 2], permutations) - PUSH
//         [1, 3, 2]
//         [1, 2, 3]
//         [1, 2, 3]
//
//         j = 1
//         [2, 1, 3]
//         permutationsHelper(1, [2, 1, 3], permutations)
//           j = 1
//           [2, 1, 3]
//           permutationsHelper(2, [2, 1, 3], permutations)
//             j = 2
//             [2, 1, 3]
//             permutationsHelper(3, [2, 1, 3], permutations) - PUSH
//             [2, 1, 3]
//             [2, 1, 3]
//
//             j = 2
//             [2, 3, 1]
//             permutationsHelper(2, [2, 3, 1], permutations)
//               j = 2
//               [2, 3, 1]
//               permutationsHelper(3, [2, 3, 1], permutations) - PUSH
//               [2, 3, 1]
//               [2, 1, 3]
//               [1, 2, 3]
//
//               j = 2
//               [3, 2, 1]
//               permutationsHelper(1, [3, 2, 1], permutations)
//                 j = 1
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/permutations/solution1/permutations.kt

```
//      [3, 2, 1]
//      permutationsHelper(2, [3, 2, 1], permutations)
//      j = 2
//      [3, 2, 1]
//      permutationsHelper(3, [3, 2, 1], permutations) - PUSH
//      [3, 2, 1]
//      [3, 2, 1]
//
//      j = 2
//      [3, 1, 2]
//      permutationsHelper(2, [3, 1, 2], permutations)
//      j = 2
//      [3, 1, 2]
//      permutationsHelper(3, [3, 1, 2], permutations) - PUSH
//      [3, 1, 2]
//      [3, 2, 1]
//      [1, 2, 3]

fun getPermutations(array: List<Int>): List<List<Int>> {
    val permutations = mutableListOf<List<Int>>()

    if (array.isEmpty()) return permutations

    permutationsHelper(0, array.toMutableList(), permutations)
    return permutations
}

fun permutationsHelper(i: Int, array: MutableList<Int>, permutations: MutableList<List<Int>>) {
    if (i == array.size) {
        permutations.add(array.toList())
    }

    for (j in i .. array.lastIndex) {
        swap (array, i, j)
        permutationsHelper(i + 1, array, permutations)
        swap(array, j, i)
    }
}

fun swap(array: MutableList<Int>, i: Int, j: Int) {
    array[i] = array[j].also {
        array[j] = array[i]
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/findSuccessor/solution1/findSuccessor.kt

```
package ae.medium.findSuccessor.solution1

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
    var parent: BinaryTree? = null
}

fun findSuccessor(tree: BinaryTree, node: BinaryTree): BinaryTree? {
    if (node.right != null) {
        return getLeftmostChild(node.right!!)
    }
    return getRightMostParent(node)
}

fun getLeftmostChild(node: BinaryTree): BinaryTree {
    var currentNode = node

    while (currentNode.left != null) {
        currentNode = currentNode.left!!
    }

    return currentNode
}

fun getRightMostParent(node: BinaryTree): BinaryTree? {
    var currentNode = node

    // We need the node for which our current node is in it's left subtree.
    // As long as current node is the right child of it's parent till then
    // continue going up.
    while (currentNode.parent != null && currentNode.parent!!.right == currentNode) {
        currentNode = currentNode.parent!!
    }

    // At this moment current node is the left child of it's parent
    // OR it's the root node.
    // So return the parent of the current node.
    return currentNode.parent
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/groupAnagrams/solution1/groupAnagrams.kt

```
package ae.medium.groupAnagrams.solution1

fun groupAnagrams(words: List<String>): List<List<String>> {
    if (words.isEmpty()) return listOf()

    val sortedWords = words.map { word ->
        String(word.toCharArray().sortedArray())
    }

    val indices = MutableList(words.size) { it }
    indices.sortWith(Comparator<Int> { idx1, idx2 ->
        sortedWords[idx1].compareTo(sortedWords[idx2])
    })

    val result = mutableListOf<List<String>>()
    var currentAnagramGroup = mutableListOf<String>()
    var currentAnagram = sortedWords[indices[0]]

    for (idx in indices) {
        val word = words[idx]
        val sortedWord = sortedWords[idx]

        if (sortedWord == currentAnagram) {
            currentAnagramGroup.add(word)
        } else {
            result.add(currentAnagramGroup)
            currentAnagram = sortedWord
            currentAnagramGroup = mutableListOf(word)
        }
    }

    result.add(currentAnagramGroup)
    return result
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/groupAnagrams/solution2/groupAnagrams.kt

```
package ae.medium.groupAnagrams.solution2

fun groupAnagrams(words: List<String>): List<List<String>> {
    val anagrams = mutableMapOf<String, MutableList<String>>()

    words.forEach { word ->
        val sortedWord = String(word.toCharArray().sortedArray())
        if (!anagrams.containsKey(sortedWord)) {
            anagrams[sortedWord] = mutableListOf(word)
        } else {
            anagrams[sortedWord]!!.add(word)
        }
    }

    return anagrams.values.toList()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/removelslands/solution1/removelslands.kt

```
package ae.medium.removeIslands.solution1

fun removeIslands(matrix: List<MutableList<Int>>): List<MutableList<Int>> {
    val rows = matrix.size
    val cols = matrix[0].size

    val onesConnectedToBorder = mutableListOf(rows) { mutableListOf(cols) { false } }

    for (r in 0 until rows) {
        for (c in 0 until cols) {
            val rowIsBorder = r == 0 || r == rows - 1
            val colIsBorder = c == 0 || c == cols - 1

            if (!rowIsBorder && !colIsBorder) continue

            if (matrix[r][c] != 1) continue

            findOnesConnectedToBorder(r, c, matrix, onesConnectedToBorder)
        }
    }

    for (r in 1 until matrix.lastIndex) {
        for (c in 1 until matrix[0].lastIndex) {
            if (onesConnectedToBorder[r][c]) {
                continue
            }

            matrix[r][c] = 0
        }
    }

    return matrix
}

fun findOnesConnectedToBorder(r: Int, c: Int, matrix: List<MutableList<Int>>, onesConnectedToBorder: MutableList<MutableList<Boolean>>) {
    val stack = mutableListOf<Pair<Int, Int>>()
    stack.add(r to c)

    while (stack.isNotEmpty()) {
        val (currentRow, currentCol) = stack.last()
        stack.removeAt(stack.lastIndex)

        if (onesConnectedToBorder[currentRow][currentCol]) continue

        onesConnectedToBorder[currentRow][currentCol] = true

        for (neighbor in getNeighbors(currentRow, currentCol, matrix)) {
            val (neighborRow, neighborCol) = neighbor

            if (matrix[neighborRow][neighborCol] != 1) continue

            stack.add(neighbor)
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/removelslands/solution1/removelslands.kt

```
        stack.add(neighbor)
    }
}

fun getNeighbors(r: Int, c: Int, matrix: List<MutableList<Int>>): List<Pair<Int, Int>> {
    return listOf(r - 1 to c, r to c - 1, r + 1 to c, r to c + 1).filter
    { (row, col) ->
        row >= 0 && col >= 0 && row < matrix.size && col < matrix[0].size
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/monotonicArray/solution1/monotonicArray.kt

```
package ae.medium.monotonicArray.solution1

fun isMonotonic(array: List<Int>): Boolean {
    var arrayState = State.UNDETECTED

    for (i in 1 .. array.lastIndex) {
        if (arrayState == State.UNDETECTED) {
            if (array[i] > array[i - 1]) {
                arrayState = State.INCREASING
            }

            if (array[i] < array[i - 1]) {
                arrayState = State.DECREASING
            }
        }

        if (arrayState == State.INCREASING && array[i] < array[i - 1]) {
            return false
        }

        if (arrayState == State.DECREASING && array[i] > array[i - 1]) {
            return false
        }
    }

    return true
}

enum class State {
    UNDETECTED, INCREASING, DECREASING
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/monotonicArray/solution2/monotonicArray.kt

```
package ae.medium.monotonicArray.solution2

fun isMonotonic(array: List<Int>): Boolean {
    var isIncreasing = true
    var isDecreasing = true

    for (i in 1 .. array.lastIndex) {
        if (array[i] > array[i - 1]) {
            isDecreasing = false
        }

        if (array[i] < array[i - 1]) {
            isIncreasing = false
        }
    }

    return isIncreasing || isDecreasing
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/reconstructBST/solution1/reconstructBST.kt

```
package ae.medium.reconstructBST.solution1

open class BST(value: Int, left: BST? = null, right: BST? = null) {
    var value = value
    var left = left
    var right = right
}

fun reconstructBst(preOrderTraversalValues: List<Int>): BST? {
    if (preOrderTraversalValues.isEmpty()) return null

    val currentValue = preOrderTraversalValues[0]
    var rightSubtreeRootIdx = preOrderTraversalValues.size

    for (i in 1 .. preOrderTraversalValues.lastIndex) {
        val value = preOrderTraversalValues[i]

        if (value >= currentValue) {
            rightSubtreeRootIdx = i
            break
        }
    }

    val leftSubtree = reconstructBst(preOrderTraversalValues.subList(1,
rightSubtreeRootIdx))
    val rightSubtree = reconstructBst(preOrderTraversalValues.subList(
rightSubtreeRootIdx, preOrderTraversalValues.size))

    val bst = BST(currentValue)
    bst.left = leftSubtree
    bst.right = rightSubtree

    return bst
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/reconstructBST/solution2/reconstructBST.kt

```
package ae.medium.reconstructBST.solution2

open class BST(value: Int, left: BST? = null, right: BST? = null) {
    var value = value
    var left = left
    var right = right
}

fun reconstructBst(preOrderTraversalValues: List<Int>): BST? {
    val treeInfo = TreeInfo(0)
    return reconstructBstFromRange(Integer.MIN_VALUE, Integer.MAX_VALUE,
        preOrderTraversalValues, treeInfo)!!
}

data class TreeInfo(var rootIdx: Int)

fun reconstructBstFromRange(lowerBound: Int, upperBound: Int,
    preOrderTraversalValues: List<Int>, currentSubtreeInfo: TreeInfo): BST? {
    if (currentSubtreeInfo.rootIdx == preOrderTraversalValues.size) {
        return null
    }

    val rootValue = preOrderTraversalValues[currentSubtreeInfo.rootIdx]

    if (rootValue < lowerBound || rootValue >= upperBound) {
        return null
    }

    currentSubtreeInfo.rootIdx += 1

    val leftSubtree = reconstructBstFromRange(lowerBound, rootValue,
        preOrderTraversalValues, currentSubtreeInfo)
    val rightSubtree = reconstructBstFromRange(rootValue, upperBound,
        preOrderTraversalValues, currentSubtreeInfo)

    val bst = BST(rootValue)
    bst.left = leftSubtree
    bst.right = rightSubtree

    return bst
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/spiralTraverse/solution1/spiralTraverse.kt

```
package ae.medium.spiralTraverse.solution1

fun spiralTraverse(array: List<List<Int>>): List<Int> {
    var sR = 0
    var eR = array.lastIndex

    var sC = 0
    var eC = array[0].lastIndex

    val result = mutableListOf<Int>()

    while (sR <= eR && sC <= eC) {
        for (c in sC .. eC) {
            result.add(array[sR][c])
        }

        for (r in sR + 1 .. eR) {
            result.add(array[r][eC])
        }

        for (c in eC - 1 downTo sC) {
            if (sR == eR) break
            result.add(array[eR][c])
        }

        for (r in eR - 1 downTo sR + 1) {
            if (sC == eC) break
            result.add(array[r][sC])
        }

        sR += 1
        eR -= 1
        sC += 1
        eC -= 1
    }

    return result
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/spiralTraverse/solution2/spiralTraverse.kt

```
package ae.medium.spiralTraverse.solution2

fun spiralTraverse(array: List<List<Int>>): List<Int> {
    val result = mutableListOf<Int>()
    spiralFill(array, 0, array.lastIndex, 0, array[0].lastIndex, result)
    return result
}

fun spiralFill(array: List<List<Int>>, startRow: Int, endRow: Int, startCol
: Int, endCol: Int, result: MutableList<Int>) {
    if (startRow > endRow || startCol > endCol) {
        return
    }

    for (c in startCol .. endCol) {
        result.add(array[startRow][c])
    }

    for (r in startRow + 1 .. endRow) {
        result.add(array[r][endCol])
    }

    for (c in endCol - 1 downTo startCol) {
        if (startRow == endRow) break
        result.add(array[endRow][c])
    }

    for (r in endRow - 1 downTo startRow + 1) {
        if (startCol == endCol) break
        result.add(array[r][startCol])
    }

    spiralFill(array, startRow + 1, endRow - 1, startCol + 1, endCol - 1,
result)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/taskAssignment/solution1/taskAssignment.kt

```
package ae.medium.taskAssignment.solution1

// Sort the map and assign one longest task and one shortest task
// (e.g. one from both ends) to each of the k workers.
fun taskAssignment(k: Int, tasks: List<Int>): List<List<Int>> {
    val taskPairs = mutableListOf<List<Int>>()
    val taskDurationsToIndices = getTaskDurationsToIndices(tasks)

    val sortedTasks = tasks.sorted().toMutableList()

    for (i in 0 until k) {
        val task1Duration = sortedTasks[i]
        val task1Indices = taskDurationsToIndices[task1Duration]!!
        val task1Index = task1Indices.removeAt(task1Indices.lastIndex)

        val task2Idx = sortedTasks.lastIndex - i
        val task2Duration = sortedTasks[task2Idx]
        val task2Indices = taskDurationsToIndices[task2Duration]!!
        val task2Index = task2Indices.removeAt(task2Indices.lastIndex)

        taskPairs.add(listOf(task1Index, task2Index))
    }

    return taskPairs
}

fun getTaskDurationsToIndices(tasks: List<Int>): Map<Int, MutableList<Int>>
>> {
    val taskDurationsToIndices = mutableMapOf<Int, MutableList<Int>>()

    tasks.forEachIndexed { idx, task ->
        if (!taskDurationsToIndices.containsKey(task)) {
            taskDurationsToIndices[task] = mutableListOf<Int>()
        }

        taskDurationsToIndices[task]!!.add(idx)
    }

    return taskDurationsToIndices
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/threeNumberSum/solution1/threeNumberSum.kt

```
package ae.medium.threeNumberSum.solution1

fun threeNumberSum(array: MutableList<Int>, targetSum: Int): List<List<Int>>
>> {
    val result = mutableListOf<List<Int>>()

    array.sort()

    for (i in 0 .. array.lastIndex) {
        val currentElement = array[i]
        val twoSumResults = twoSum(array, i + 1, array.lastIndex, targetSum
- currentElement)

        for (twoSumResult in twoSumResults) {
            result.add(listOf(currentElement, twoSumResult[0], twoSumResult
[1]))
        }
    }

    return result
}

fun twoSum(array: List<Int>, left: Int, right: Int, target: Int): List<List<Int>> {
    val result = mutableListOf<List<Int>>()

    var leftIdx = left
    var rightIdx = right

    while (leftIdx < rightIdx) {
        val sum = array[leftIdx] + array[rightIdx]

        if (sum == target) {
            result.add(listOf(array[leftIdx], array[rightIdx])))
            leftIdx += 1
            rightIdx -= 1
        } else if (sum < target) {
            leftIdx += 1
        } else {
            rightIdx -= 1
        }
    }

    return result
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/arrayOfProducts/solution1/arrayOfProducts.kt

```
package ae.medium.arrayOfProducts.solution1

fun arrayOfProducts(array: List<Int>): List<Int> {

    val leftProducts = MutableList<Int>(array.size) { 0 }
    val rightProducts = MutableList<Int>(array.size) { 0 }

    var leftRunningProduct = 1
    for (i in 0 .. array.lastIndex) {
        leftProducts[i] = leftRunningProduct
        leftRunningProduct *= array[i]
    }

    var rightRunningProduct = 1
    for (i in array.lastIndex downTo 0) {
        rightProducts[i] = rightRunningProduct
        rightRunningProduct *= array[i]
    }

    return leftProducts.mapIndexed { idx, leftProduct ->
        leftProduct * rightProducts[idx]
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/arrayOfProducts/solution2/arrayOfProducts.kt

```
package ae.medium.arrayOfProducts.solution2

fun arrayOfProducts(array: List<Int>): List<Int> {
    val products = MutableList(array.size) { 1 }

    var leftRunningProduct = 1
    for (i in 0 .. array.lastIndex) {
        products[i] = leftRunningProduct
        leftRunningProduct *= array[i]
    }

    var rightRunningProduct = 1
    for (i in array.lastIndex downTo 0) {
        products[i] *= rightRunningProduct
        rightRunningProduct *= array[i]
    }

    return products
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/bstConstruction/solution1/bstConstruction.kt

```
package ae.medium.bstConstruction.solution1
```

```
open class BST(value: Int) {  
    var value = value  
    var left: BST? = null  
    var right: BST? = null  
  
    fun insert(value: Int): BST {  
        if (value < this.value) {  
            if (left == null) {  
                left = BST(value)  
            } else {  
                left!!.insert(value)  
            }  
        } else {  
            if (right == null) {  
                right = BST(value)  
            } else {  
                right!!.insert(value)  
            }  
        }  
        return this  
    }  
  
    fun contains(value: Int): Boolean {  
        return if (value < this.value) {  
            left?.contains(value) ?: false  
        } else if (value > this.value) {  
            right?.contains(value) ?: false  
        } else {  
            true  
        }  
    }  
  
    fun remove(value: Int, parent: BST? = null): BST {  
        if (value < this.value) {  
            left?.remove(value, this)  
        } else if (value > this.value) {  
            right?.remove(value, this)  
        } else {  
            if (left != null && right != null) {  
                this.value = right!!.getMinValue()  
                right!!.remove(this.value, this)  
            } else if (parent == null) {  
                if (left != null) {  
                    this.value = left!!.value  
                    right = left!!.right  
                    left = left!!.left  
                } else if (right != null) {  
                    this.value = right!!.value  
                    left = right!!.left  
                }  
            } else {  
                if (parent.left == this) {  
                    parent.left = right  
                } else {  
                    parent.right = left  
                }  
            }  
        }  
        return this  
    }  
}
```

```
File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/bstConstruction/solution1/bstConstruction.kt

        right = right!!.right
    } else {
        // Node has no left child and no right child so how to
remove it
    }
} else if (parent.left == this) {
    parent.left = if (left != null) left else right
} else if (parent.right == this) {
    parent.right = if (left != null) left else right
}
}

return this
}

fun getMinValue(): Int {
    return if (left != null) {
        left!!.getMinValue()
    } else this.value
}
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/bstConstruction/solution2/bstConstruction.kt

```
package ae.medium.bstConstruction.solution2

open class BST(value: Int) {
    var value = value
    var left: BST? = null
    var right: BST? = null

    fun insert(value: Int): BST {
        var currentNode: BST? = this

        while (true) {
            if (value < currentNode!!.value) {
                if (currentNode.left == null) {
                    currentNode.left = BST(value)
                    break
                } else {
                    currentNode = currentNode.left
                }
            } else {
                if (currentNode.right == null) {
                    currentNode.right = BST(value)
                    break
                } else {
                    currentNode = currentNode.right
                }
            }
        }
    }

    return this
}

fun contains(value: Int): Boolean {
    var currentNode: BST? = this

    while (currentNode != null) {
        if (value < currentNode.value) {
            currentNode = currentNode.left
        } else if (value > currentNode.value) {
            currentNode = currentNode.right
        } else {
            return true
        }
    }
}

return false
}

fun remove(value: Int, parent: BST? = null): BST {
    var currentNode: BST? = this
    var parentNode = parent

    while (currentNode != null) {
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/bstConstruction/solution2/bstConstruction.kt

```
    if (value < currentNode.value) {
        parentNode = currentNode
        currentNode = currentNode.left
    } else if (value > currentNode.value) {
        parentNode = currentNode
        currentNode = currentNode.right
    } else {
        if (currentNode.left != null && currentNode.right != null
) {
            currentNode.value = currentNode.right!!.getMinValue()
            currentNode.right!!.remove(currentNode.value,
currentNode)
        } else if (parentNode == null) {
            if (currentNode.left != null) {
                currentNode.value = currentNode.left!!.value
                currentNode.right = currentNode.left!!.right
                currentNode.left = currentNode.left!!.left
            } else if (currentNode.right != null) {
                currentNode.value = currentNode.right!!.value
                currentNode.left = currentNode.right!!.left
                currentNode.right = currentNode.right!!.right
            } else {
                // Node has no left child and no right child so how
to remove it
            }
        } else if (parentNode.left == currentNode) {
            parentNode.left = if (currentNode.left != null)
currentNode.left else currentNode.right
        } else if (parentNode.right == currentNode) {
            parentNode.right = if (currentNode.left != null)
currentNode.left else currentNode.right
        }
    }
    break
}
}

return this
}

fun getMinValue(): Int {
    var currentNode: BST? = this

    while (currentNode?.left != null) {
        currentNode = currentNode.left
    }

    return currentNode!!.value
}
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/threeNumberSort/solution1/threeNumberSort.kt

```
package ae.medium.threeNumberSort.solution1

fun threeNumberSort(array: MutableList<Int>, order: List<Int>): List<Int> {
    val valueCounts = mutableListOf(0, 0, 0)
    val sortedArray = MutableList(array.size) { 0 }

    for (i in array) {
        val orderIndex = order.indexOf(i)
        valueCounts[orderIndex] += 1
    }

    var arrayIdx = 0

    for (i in 0 until 3) {
        val item = order[i]
        val count = valueCounts[i]

        for (j in 0 until count) {
            sortedArray[arrayIdx] = item
            arrayIdx += 1
        }
    }

    return sortedArray
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/threeNumberSort/solution2/threeNumberSort.kt

```
package ae.medium.threeNumberSort.solution2

fun threeNumberSort(array: MutableList<Int>, order: List<Int>): List<Int> {
    val mutableArray = array.toMutableList()

    val (firstValue, _, thirdValue) = order

    var firstIdx = 0
    for (i in 0 .. mutableArray.lastIndex) {
        if (mutableArray[i] == firstValue) {
            swap(mutableArray, i, firstIdx)
            firstIdx += 1
        }
    }

    var thirdIdx = mutableArray.lastIndex
    for (i in mutableArray.lastIndex downTo 0) {
        if (mutableArray[i] == thirdValue) {
            swap(mutableArray, i, thirdIdx)
            thirdIdx -= 1
        }
    }

    return mutableArray
}

fun swap(array: MutableList<Int>, i: Int, j: Int) {
    array[i] = array[j].also {
        array[j] = array[i]
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/threeNumberSort/solution3/threeNumberSort.kt

```
package ae.medium.threeNumberSort.solution3

fun threeNumberSort(array: MutableList<Int>, order: List<Int>): List<Int> {

    val mutableArray = array.toMutableList()

    val (firstValue, secondValue, _) = order
    var firstIdx = 0
    var secondIdx = 0
    var thirdIdx = array.lastIndex

    while (secondIdx <= thirdIdx) {
        if (mutableArray[secondIdx] == firstValue) {
            swap(mutableArray, secondIdx, firstIdx)
            firstIdx += 1
            secondIdx += 1
        } else if (mutableArray[secondIdx] == secondValue) {
            secondIdx += 1
        } else {
            swap(mutableArray, secondIdx, thirdIdx)
            thirdIdx -= 1
        }
    }

    return mutableArray
}

fun swap(array: MutableList<Int>, i: Int, j: Int) {
    array[i] = array[j].also {
        array[j] = array[i]
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/balancedBrackets/solution1/balancedBrackets.kt

```
package ae.medium.balancedBrackets.solution1

import java.util.Stack

fun balancedBrackets(str: String): Boolean {

    val openingBrackets = "[{("
    val closingBrackets = "])})"

    val matchingBrackets = mapOf(
        ')' to '(',
        '}' to '{',
        ']' to '['
    )

    val stack = Stack<Char>()
    for (c in str) {
        if (openingBrackets.contains(c)) {
            stack.push(c)
        } else if (closingBrackets.contains(c)) {
            if (stack.isEmpty()) {
                return false
            } else if (matchingBrackets[c]!! != stack.peek()) {
                stack.pop()
            } else {
                return false
            }
        }
    }

    return stack.isEmpty()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/invertBinaryTree/solution1/invertBinaryTree.kt

```
package ae.medium.invertBinaryTree.solution1

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun invertBinaryTree(tree: BinaryTree) {
    invertBinaryTreeHelper(tree)
}

fun invertBinaryTreeHelper(tree: BinaryTree?) {
    tree ?: return

    swapLeftAndRight(tree)

    invertBinaryTreeHelper(tree.left)
    invertBinaryTreeHelper(tree.right)
}

fun swapLeftAndRight(tree: BinaryTree) {
    tree.left = tree.right.also {
        tree.right = tree.left
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/invertBinaryTree/solution2/invertBinaryTree.kt

```
package ae.medium.invertBinaryTree.solution2
```

```
import java.util.*
```

```
open class BinaryTree(value: Int) {  
    var value = value  
    var left: BinaryTree? = null  
    var right: BinaryTree? = null  
}  
  
fun invertBinaryTree(tree: BinaryTree) {  
    val queue = LinkedList<BinaryTree>()  
    queue.add(tree)  
  
    while (queue.isNotEmpty()) {  
        val current = queue.pollFirst()  
  
        current.left = current.right.also {  
            current.right = current.left  
        }  
  
        if (current.left != null) {  
            queue.add(current.left!!)  
        }  
        if (current.right != null) {  
            queue.add(current.right!!)  
        }  
    }  
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/kadanesAlgorithm/solution1/kadanesAlgorithm.kt

```
package ae.medium.kadanesAlgorithm.solution1
```

```
import kotlin.math.max
```

```
// maxEndingHere = 3, maxSoFar = 3
// 3, 5, -9, 1, 3, -2, 3, 4, 7, 2, -9, 6, 3, 1, -5, 4
// 5 - maxEndingHere = 8, maxSoFar = 8
// -9 - maxEndingHere = -1, maxSoFar = 8
// 1 - maxEndingHere = 1, maxSoFar = 8
// 3 - maxEndingHere = 4, maxSoFar = 8
// -2 - maxEndingHere = 2, maxSoFar = 8
// 3 - maxEndingHere = 5, maxSoFar = 8
// 4 - maxEndingHere = 9, maxSoFar = 9
// 7 - maxEndingHere = 16, maxSoFar = 16
// 2 - maxEndingHere = 18, maxSoFar = 18
// -9 - maxEndingHere = 9, maxSoFar = 18
// 6 - maxEndingHere = 15, maxSoFar = 18
// 3 - maxEndingHere = 18, maxSoFar = 18
// 1 - maxEndingHere = 19, maxSoFar = 19
// -5 - maxEndingHere = 14, maxSoFar = 19
// 4 - maxEndingHere = 18, maxSoFar = 19
// return 19
```

```
fun kadanesAlgorithm(array: List<Int>): Int {
```

```
    var maxEndingHere = array[0]
```

```
    var maxSoFar = array[0]
```

```
        for (i in 1 .. array.lastIndex) {
```

```
            maxEndingHere = max(array[i], maxEndingHere + array[i])
```

```
            maxSoFar = max(maxEndingHere, maxSoFar)
```

```
}
```

```
    return maxSoFar
```

```
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/moveElementToEnd/solution1/moveElementToEnd.kt

```
package ae.medium.moveElementToEnd.solution1

fun moveElementToEnd(array: MutableList<Int>, toMove: Int): List<Int> {
    var slow = 0
    var fast = 0

    while (fast < array.size) {
        while (fast < array.size && array[fast] == toMove) {
            fast += 1
        }

        if (fast < array.size) {
            array[slow] = array[fast]
            slow += 1
            fast += 1
        }
    }

    for (i in slow .. array.lastIndex) {
        array[i] = toMove
    }

    return array
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/moveElementToEnd/solution2/moveElementToEnd.kt

```
package ae.medium.moveElementToEnd.solution2

fun moveElementToEnd(array: MutableList<Int>, toMove: Int): List<Int> {
    var i = 0
    var j = array.lastIndex

    while (i < j) {
        while (i < j && array[j] == toMove) {
            j -= 1
        }

        if (array[i] == toMove) {
            swap(i, j, array)
        }

        i += 1
    }

    return array
}

fun swap(i: Int, j: Int, array: MutableList<Int>) {
    array[i] = array[j].also {
        array[j] = array[i]
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/singleCycleCheck/solution1/singleCycleCheck.kt

```
package ae.medium.singleCycleCheck.solution1

fun hasSingleCycle(array: List<Int>): Boolean {
    var numElementsVisited = 0

    // We should start checking from index 0
    var currentIdx = 0

    // Until we visit ALL the elements
    while (numElementsVisited < array.size) {

        // If we visited some elements (not all of them - imposed by the
        enclosing while loop)
        // But we still came back to the starting index 0
        if (numElementsVisited > 0 && currentIdx == 0)
            return false

        numElementsVisited += 1

        // Get next index and set as currentIdx
        currentIdx = getNextIdx(array, currentIdx)
    }

    // Once the while loop breaks we visited all elements
    // Now to say that if we have a single cycle we need to check if we are
    back at starting index 0.
    return currentIdx == 0
}

fun getNextIdx(array: List<Int>, currentIdx: Int): Int {
    val arrayLen = array.size
    val jumpValue = array[currentIdx]

    // To roll over at an index within bound we use modulus by array length
    val nextIdx = (currentIdx + jumpValue) % arrayLen

    // If nextIdx is negative (can occur for some negative element)
    // we need to bring them back into the bound by adding arrayLen
    return if (nextIdx >= 0) nextIdx else nextIdx + arrayLen
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/sumOfLinkedLists/solution1/sumOfLinkedLists.kt

```
package ae.medium.sumOfLinkedLists.solution1

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun sumOfLinkedLists(linkedListOne: LinkedList, linkedListTwo: LinkedList): LinkedList {
    val preHead = LinkedList(-1)

    var currentNode = preHead
    var carry = 0

    var linkedListOneCurrent: LinkedList? = linkedListOne
    var linkedListTwoCurrent: LinkedList? = linkedListTwo

    while (linkedListOneCurrent != null || linkedListTwoCurrent != null || carry > 0) {
        val valueOne = linkedListOneCurrent?.value ?: 0
        val valueTwo = linkedListTwoCurrent?.value ?: 0
        val sumOfNodes = valueOne + valueTwo + carry

        val newValue = sumOfNodes % 10
        val newNode = LinkedList(newValue)
        currentNode.next = newNode
        currentNode = newNode

        carry = sumOfNodes / 10

        linkedListOneCurrent = linkedListOneCurrent?.next
        linkedListTwoCurrent = linkedListTwoCurrent?.next
    }

    return preHead.next!!
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/validIpAddresses/solution1/validIpAddresses.kt

```
package ae.medium.validIpAddresses.solution1

import kotlin.math.min

fun validIPAddresses(string: String): List<String> {
    val ipAddresses = mutableListOf<String>()

    for (i in 1 until min(string.length, 4)) {
        val ipAddressParts = MutableList(4) { "" }

        ipAddressParts[0] = string.substring(0, i)
        if (!isValidPart(ipAddressParts[0])) {
            continue
        }

        for (j in i + 1 until min(i + 4, string.length)) {
            ipAddressParts[1] = string.substring(i, j)
            if (!isValidPart(ipAddressParts[1])) {
                continue
            }

            for (k in j + 1 until min(j + 4, string.length)) {
                ipAddressParts[2] = string.substring(j, k)
                ipAddressParts[3] = string.substring(k)

                if (!isValidPart(ipAddressParts[2]) || !isValidPart(
                    ipAddressParts[3])) {
                    continue
                }

                ipAddresses.add(ipAddressParts.joinToString("."))
            }
        }
    }

    return ipAddresses
}

fun isValidPart(str: String): Boolean {
    val strAsInt = str.toInt()
    if (strAsInt > 255) {
        return false
    }
    return str.length == strAsInt.toString().length
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/validStartingCity/solution1/validStartingCity.kt

```
package ae.medium.validStartingCity.solution1

fun validStartingCity(distances: List<Int>, fuel: List<Int>, mpg: Int): Int {
    val numCities = distances.size

    for (startingCityIdx in 0 until numCities) {
        var milesRemaining = 0

        for (currentCityIdx in startingCityIdx until startingCityIdx + numCities) {
            val currentCityIdxTranslated = currentCityIdx % numCities

            val fuelInCurrentCity = fuel[currentCityIdxTranslated]
            val distanceToNextCity = distances[currentCityIdxTranslated]

            milesRemaining += fuelInCurrentCity * mpg - distanceToNextCity

            if (milesRemaining < 0) {
                break
            }
        }

        if (milesRemaining >= 0) {
            return startingCityIdx
        }
    }

    return -1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/validStartingCity/solution2/validStartingCity.kt

```
package ae.medium.validStartingCity.solution2

fun validStartingCity(distances: List<Int>, fuel: List<Int>, mpg: Int): Int {
    val numCities = distances.size

    var startingCityCandidate = 0
    var milesRemainingInStartingCityCandidate = 0

    var arrivedToCurrentCityWithMiles = 0

    for (currentCityIdx in 1 until numCities) {
        val fuelInPreviousCity = fuel[currentCityIdx - 1]
        val distanceFromPreviousCity = distances[currentCityIdx - 1]

        arrivedToCurrentCityWithMiles += fuelInPreviousCity * mpg -
distanceFromPreviousCity

        if (arrivedToCurrentCityWithMiles <
milesRemainingInStartingCityCandidate) {
            milesRemainingInStartingCityCandidate =
arrivedToCurrentCityWithMiles
            startingCityCandidate = currentCityIdx
        }
    }

    return startingCityCandidate
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/binaryTreeDiameter/solution1/binaryTreeDiameter.kt

```
package ae.medium.binaryTreeDiameter.solution1

import kotlin.math.max

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun binaryTreeDiameter(tree: BinaryTree?): Int {
    return getTreeInfo(tree).diameter
}

data class TreeInfo(var diameter: Int, var height: Int)

fun getTreeInfo(tree: BinaryTree?): TreeInfo {
    if (tree == null) {
        return TreeInfo(0, 0)
    }

    val leftTreeInfo = getTreeInfo(tree.left)
    val rightTreeInfo = getTreeInfo(tree.right)

    val longestPathThroughRoot = leftTreeInfo.height + rightTreeInfo.height
    val maxDiameterSoFar = max(leftTreeInfo.diameter, rightTreeInfo.
diameter)
    val currentDiameter = max(longestPathThroughRoot, maxDiameterSoFar)
    val currentHeight = 1 + max(leftTreeInfo.height, rightTreeInfo.height)

    return TreeInfo(currentDiameter, currentHeight)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/breadthFirstSearch/solution1/breadthFirstSearch.kt

```
package ae.medium.breadthFirstSearch.solution1

import java.util.LinkedList

class Node(name: String) {
    val name: String = name
    val children = mutableListOf<Node>()

    fun breadthFirstSearch(): List<String> {
        val array = mutableListOf<String>()

        val queue = LinkedList<Node>()
        queue.add(this)

        while (queue.isNotEmpty()) {
            val currentNode = queue.poll()
            array.add(currentNode.name)

            for (child in currentNode.children) {
                queue.add(child)
            }
        }

        return array
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/nextGreaterElement/solution1/nextGreaterElement.kt

```
package ae.medium.nextGreaterElement.solution1

import java.util.Stack

fun nextGreaterElement(array: MutableList<Int>): MutableList<Int> {
    val result = MutableList(array.size) { -1 }
    val stack = Stack<Int>()

    for (i in 0 until 2 * array.size) {
        val circularIdx = i % array.size

        while (stack.isNotEmpty() && array[circularIdx] > array[stack.peek()])
        {
            val top = stack.pop()
            result[top] = array[circularIdx]
        }

        stack.push(circularIdx)
    }

    return result
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/nextGreaterElement/solution2/nextGreaterElement.kt

```
package ae.medium.nextGreaterElement.solution2

import java.util.Stack

fun nextGreaterElement(array: MutableList<Int>): MutableList<Int> {
    val result = MutableList(array.size) { -1 }
    val stack = Stack<Int>()

    for (i in (2 * array.size - 1) downTo 0) {
        val circularIdx = i % array.size

        while (stack.isNotEmpty()) {
            if (stack.peek() <= array[circularIdx]) {
                stack.pop()
            } else {
                result[circularIdx] = stack.peek()
                break
            }
        }

        stack.push(array[circularIdx])
    }

    return result
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/smallestDifference/solution1/smallestDifference.kt

```
package ae.medium.smallestDifference.solution1

import kotlin.math.abs

fun smallestDifference(arrayOne: MutableList<Int>, arrayTwo: MutableList<Int>): List<Int> {
    arrayOne.sort()
    arrayTwo.sort()

    var i = 0
    var j = 0

    var minDiff = Integer.MAX_VALUE
    var minPair: List<Int> = listOf<Int>()

    while (i < arrayOne.size && j < arrayTwo.size) {
        val numOne = arrayOne[i]
        val numTwo = arrayTwo[j]

        if (numOne < numTwo) {
            i += 1
        } else if (numTwo < numOne) {
            j += 1
        } else {
            return listOf(numOne, numTwo)
        }

        if (abs(numOne - numTwo) < minDiff) {
            minDiff = abs(numOne - numTwo)
            minPair = listOf(numOne, numTwo)
        }
    }

    return minPair
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/staircaseTraversal/solution1/staircaseTraversal.kt

```
package ae.medium.staircaseTraversal.solution1

import kotlin.math.min

// O(n * k) time | O(n) space
fun staircaseTraversal(height: Int, maxSteps: Int): Int {
    val memo = mutableMapOf(0 to 1, 1 to 1)
    return numberOfWorksToTop(height, maxSteps, memo)
}

fun numberOfWorksToTop(height: Int, maxSteps: Int, memo: MutableMap<Int, Int>): Int {
    if (memo.containsKey(height)) {
        return memo[height]!!
    }

    var numberOfWays = 0
    for (step in 1 .. min(height, maxSteps)) {
        numberOfWays += numberOfWorksToTop(height - step, maxSteps, memo)
    }

    memo[height] = numberOfWays
    return memo[height]!!
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/staircaseTraversal/solution2/staircaseTraversal.kt

```
package ae.medium.staircaseTraversal.solution2

import kotlin.math.min

// O(n * k) time | O(n) space
fun staircaseTraversal(height: Int, maxSteps: Int): Int {
    val waysToTop = mutableListOf(height + 1) { 0 }
    waysToTop[0] = 1
    waysToTop[1] = 1

    for (currentHeight in 2 .. height) {

        for (step in 1 .. min(currentHeight, maxSteps)) {
            waysToTop[currentHeight] += waysToTop[currentHeight - step]
        }
    }

    return waysToTop.last()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/staircaseTraversal/solution3/staircaseTraversal.kt

```
package ae.medium.staircaseTraversal.solution3

// O(n) time | O(n) space
fun staircaseTraversal(height: Int, maxSteps: Int): Int {
    var currentNumberOfWays = 0
    val waysToTop = mutableListOf(1)

    for (currentHeight in 1 .. height) {
        val startOfWindow = currentHeight - maxSteps - 1
        val endOfWindow = currentHeight - 1

        if (startOfWindow >= 0) {
            currentNumberOfWays -= waysToTop[startOfWindow]
        }

        currentNumberOfWays += waysToTop[endOfWindow]
        waysToTop.add(currentNumberOfWays)
    }

    return waysToTop.last()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/firstDuplicateValue/solution1/firstDuplicateValue.kt

```
package ae.medium.firstDuplicateValue.solution1

fun firstDuplicateValue(array: MutableList<Int>): Int {
    val seen = mutableSetOf<Int>()

    for (i in array) {
        if (seen.contains(i)) {
            return i
        }

        seen.add(i)
    }

    return -1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/firstDuplicateValue/solution2/firstDuplicateValue.kt

```
package ae.medium.firstDuplicateValue.solution2

import kotlin.math.abs

//      [ 2.  1,  5,  4,  3,  3,  4]
// 2 - [ 2, -1,  5,  4,  3,  3,  4]
// 1 - [-2, -1,  5,  4,  3,  3,  4]
// 5 - [-2, -1,  5,  4, -3,  3,  4]
// 4 - [-2, -1,  5, -4, -3,  3,  4]
// -3 - mappedIndex = abs(-3) - 1 = 2
//          (array[2] > 0)
//          array[2] *= -1
//      [-2, -1, -5, -4, -3,  3,  4]
// 3 - mappedIndex = abs(3) - 1 = 2
//          (array[2] < 0)
//          return abs(3) = 3
fun firstDuplicateValue(array: MutableList<Int>): Int {

    for (i in array) {
        val mappedIndex = abs(i) - 1

        if (array[mappedIndex] < 0) {
            return abs(i)
        }

        array[mappedIndex] *= -1
    }

    return -1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/levenshteinDistance/solution1/levenshteinDistance.kt

```
package ae.medium.levenshteinDistance.solution1

import kotlin.math.min

fun levenshteinDistance(str1: String, str2: String): Int {
    val edits = MutableList(str1.length + 1) { MutableList(str2.length + 1) { 0 } }

    for (i in 0 .. str1.length) {
        edits[i][0] = i
    }

    for (j in 0 .. str2.length) {
        edits[0][j] = j
    }

    // If i, j pair characters are equal then take the diagonal
    // else take the min of three neighboring numbers + 1
    for (i in 1 .. str1.length) {
        for (j in 1 .. str2.length) {
            if (str1[i - 1] == str2[j - 1]) {
                edits[i][j] = edits[i - 1][j - 1]
            } else {
                edits[i][j] = 1 + min(
                    min(edits[i - 1][j], edits[i][j - 1]),
                    edits[i - 1][j - 1]
                )
            }
        }
    }

    return edits[str1.length][str2.length]
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minHeapConstruction/solution1/minHeapConstruction.kt

```
package ae.medium.minHeapConstruction.solution1

open class MinHeap(array: MutableList<Int>) {
    val heap = this.buildHeap(array)

    fun buildHeap(array: MutableList<Int>): MutableList<Int> {
        val firstParentIdx = (array.lastIndex - 1) / 2
        var parentIdx = firstParentIdx

        while (parentIdx >= 0) {
            siftDown(parentIdx, array.lastIndex, array)
            parentIdx -= 1
        }

        return array
    }

    fun siftDown(currentIdx: Int, endIdx: Int, heap: MutableList<Int>) {
        var mutableCurrentIdx = currentIdx

        var childOneIdx = mutableCurrentIdx * 2 + 1

        while (childOneIdx <= endIdx) {
            val childTwoIdx = if (mutableCurrentIdx * 2 + 2 <= endIdx)
                mutableCurrentIdx * 2 + 2 else -1
            val idxToSwap = if (childTwoIdx != -1 && heap[childTwoIdx] <
                heap[childOneIdx]) {
                childTwoIdx
            } else childOneIdx

            if (heap[idxToSwap] < heap[mutableCurrentIdx]) {
                swap(mutableCurrentIdx, idxToSwap, heap)
                mutableCurrentIdx = idxToSwap
                childOneIdx = mutableCurrentIdx * 2 + 1
            } else {
                // Already in the correct position
                return
            }
        }
    }

    fun siftUp(currentIdx: Int, heap: MutableList<Int>) {
        var mutableCurrentIdx = currentIdx

        var parentIdx = (mutableCurrentIdx - 1) / 2

        while (mutableCurrentIdx > 0 && heap[mutableCurrentIdx] < heap[
            parentIdx]) {
            swap(mutableCurrentIdx, parentIdx, heap)
            mutableCurrentIdx = parentIdx
            parentIdx = (mutableCurrentIdx - 1) / 2
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minHeapConstruction/solution1/minHeapConstruction.kt

```
}

fun peek(): Int? {
    return heap[0]
}

fun remove(): Int? {
    swap(heap.lastIndex, 0, heap)
    val nodeToRemove = heap.removeAt(heap.lastIndex)
    siftDown(0, heap.lastIndex, heap)
    return nodeToRemove
}

fun insert(value: Int) {
    heap.add(value)
    siftUp(heap.lastIndex, heap)
}

fun swap(i: Int, j: Int, heap: MutableList<Int>) {
    val temp = heap[i]
    heap[i] = heap[j]
    heap[j] = temp
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/phoneNumberMnemonics/solution1/phoneNumberMnemonics.kt

```
package ae.medium.phoneNumberMnemonics.solution1

fun phoneNumberMnemonics(phoneNumber: String): List<String> {
    val keyMap = mapOf(
        0 to "0",
        1 to "1",
        2 to "abc",
        3 to "def",
        4 to "ghi",
        5 to "jkl",
        6 to "mno",
        7 to "pqrs",
        8 to "tuv",
        9 to "wxyz"
    )
    val mnemonics = mutableListOf<String>()
    helper(phoneNumber, mnemonics, keyMap, mutableListOf(), 0)

    return mnemonics
}

fun helper(phoneNumber: String, mnemonics: MutableList<String>, keyMap: Map<Int, String>, current: MutableList<Char>, idx: Int) {
    if (idx == phoneNumber.length) {
        mnemonics.add(String(current.toCharArray()))
        return
    }

    val keyChars = keyMap[phoneNumber[idx] - '0']!!
    for (c in keyChars) {
        current.add(c)
        helper(phoneNumber, mnemonics, keyMap, current, idx + 1)
        current.removeAt(current.lastIndex)
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/removeKthNodeFromEnd/solution1/removeKthNodeFromEnd.kt

```
package ae.medium.removeKthNodeFromEnd.solution1

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun removeKthNodeFromEnd(head: LinkedList, k: Int) {
    var first: LinkedList? = head
    var second: LinkedList? = head

    var counter = 1

    while (second != null && counter <= k) {
        second = second.next
        counter += 1
    }

    if (second == null) {
        head.value = head.next!!.value
        head.next = head.next!!.next
        return
    }

    while (second?.next != null) {
        first = first!!.next
        second = second.next
    }

    first!!.next = first.next!!.next
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/reverseWordsInString/solution1/reverseWordsInString.kt

```
package ae.medium.reverseWordsInString.solution1

fun reverseWordsInString(string: String): String {
    val words = mutableListOf<String>()
    var start = 0

    for (idx in string.indices) {
        val ch = string[idx]

        if (ch == ' ') {
            words.add(string.substring(start, idx))
            start = idx
        } else if (string[start] == ' ') {
            words.add(" ")
            start = idx
        }
    }

    words.add(string.substring(start))

    words.reverse()

    return words.joinToString("")
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/reverseWordsInString/solution2/reverseWordsInString.kt

```
package ae.medium.reverseWordsInString.solution2

fun reverseWordsInString(string: String): String {
    val charList = string.toCharArray().toMutableList()
    reverseSegment(charList, 0, charList.lastIndex)

    var startIdx = 0
    var endIdx = 0

    while (endIdx < charList.size) {
        endIdx = startIdx

        while (endIdx < charList.size && charList[endIdx] != ' ') {
            endIdx += 1
        }

        reverseSegment(charList, startIdx, endIdx - 1)

        startIdx = endIdx + 1
    }

    return String(charList.toCharArray())
}

fun reverseSegment(charList: MutableList<Char>, left: Int, right: Int) {
    var leftIdx = left
    var rightIdx = right

    while (leftIdx < rightIdx) {
        charList[leftIdx] = charList[rightIdx].also {
            charList[rightIdx] = charList[leftIdx]
        }

        leftIdx += 1
        rightIdx -= 1
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/searchInSortedMatrix/solution1/searchInSortedMatrix.kt

```
package ae.medium.searchInSortedMatrix.solution1

// O(n + m) time | O(1) space
fun searchInSortedMatrix(matrix: List<List<Int>>, target: Int): Pair<Int, Int> {
    // Start from top-right
    var r = 0
    var c = matrix[0].lastIndex

    while (r < matrix.size && c >= 0) {
        if (matrix[r][c] < target) {
            r += 1
        } else if (matrix[r][c] > target) {
            c -= 1
        } else {
            return r to c
        }
    }

    return -1 to -1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minimumPassesOfMatrix/solution1/minimumPassesOfMatrix.kt

```
package ae.medium.minimumPassesOfMatrix.solution1

import java.util.LinkedList

fun minimumPassesOfMatrix(matrix: MutableList<MutableList<Int>>): Int {
    val passes = convertNegatives(matrix)
    return if (isAllPositive(matrix)) passes - 1 else -1
}

fun convertNegatives(matrix: MutableList<MutableList<Int>>): Int {
    var nextPassQueue = getPositiveIndices(matrix)

    var passes = 0

    while (nextPassQueue.isNotEmpty()) {
        val currentPassQueue = nextPassQueue
        nextPassQueue = LinkedList<List<Int>>()

        while (currentPassQueue.isNotEmpty()) {
            val (currentRow, currentCol) = currentPassQueue.poll()
            val neighbors = getNeighbors(currentRow, currentCol, matrix)

            for (neighbor in neighbors) {
                val (neighborR, neighborC) = neighbor

                if (matrix[neighborR][neighborC] < 0) {
                    matrix[neighborR][neighborC] *= -1
                    nextPassQueue.add(neighbor)
                }
            }
        }

        passes += 1
    }

    return passes
}

fun getPositiveIndices(matrix: MutableList<MutableList<Int>>): LinkedList<List<Int>> {
    val queue = LinkedList<List<Int>>()

    for (r in 0 until matrix.size) {
        for (c in 0 until matrix[0].size) {
            if (matrix[r][c] > 0) {
                queue.add(listOf(r, c))
            }
        }
    }

    return queue
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minimumPassesOfMatrix/solution1/minimumPassesOfMatrix.kt

```
fun getNeighbors(currentRow: Int, currentCol: Int, matrix: MutableList<MutableList<Int>>): List<List<Int>> {
    val neighbors = mutableListOf(listOf(currentRow - 1, currentCol),
    listOf(currentRow, currentCol - 1), listOf(currentRow + 1, currentCol),
    listOf(currentRow, currentCol + 1))
    return neighbors.filter { (r, c) ->
        r >= 0 && r < matrix.size && c >= 0 && c < matrix[0].size
    }
}

fun isAllPositive(matrix: MutableList<MutableList<Int>>): Boolean {
    for (r in 0 until matrix.size) {
        for (c in 0 until matrix[0].size) {
            if (matrix[r][c] < 0) {
                return false
            }
        }
    }
    return true
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minimumPassesOfMatrix/solution2/minimumPassesOfMatrix.kt

```
package ae.medium.minimumPassesOfMatrix.solution2

import java.util.LinkedList

fun minimumPassesOfMatrix(matrix: MutableList<MutableList<Int>>): Int {
    val passes = convertNegatives(matrix)
    return if (isAllPositive(matrix)) passes - 1 else -1
}

fun convertNegatives(matrix: MutableList<MutableList<Int>>): Int {
    var queue = getPositiveIndices(matrix)

    var passes = 0

    while (queue.isNotEmpty()) {
        var currentSize = queue.size

        while (currentSize > 0) {
            val (currentRow, currentCol) = queue.poll()
            val neighbors = getNeighbors(currentRow, currentCol, matrix)

            for (neighbor in neighbors) {
                val (neighborR, neighborC) = neighbor

                if (matrix[neighborR][neighborC] < 0) {
                    matrix[neighborR][neighborC] *= -1
                    queue.add(neighbor)
                }
            }
            currentSize -= 1
        }

        passes += 1
    }

    return passes
}

fun getPositiveIndices(matrix: MutableList<MutableList<Int>>): LinkedList<List<Int>> {
    val queue = LinkedList<List<Int>>()

    for (r in 0 until matrix.size) {
        for (c in 0 until matrix[0].size) {
            if (matrix[r][c] > 0) {
                queue.add(listOf(r, c))
            }
        }
    }

    return queue
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minimumPassesOfMatrix/solution2/minimumPassesOfMatrix.kt

```
fun getNeighbors(currentRow: Int, currentCol: Int, matrix: MutableList<MutableList<Int>>): List<List<Int>> {
    val neighbors = mutableListOf(listOf(currentRow - 1, currentCol),
listOf(currentRow, currentCol - 1), listOf(currentRow + 1, currentCol),
listOf(currentRow, currentCol + 1))
    return neighbors.filter { (r, c) ->
        r >= 0 && r < matrix.size && c >= 0 && c < matrix[0].size
    }
}

fun isAllPositive(matrix: MutableList<MutableList<Int>>): Boolean {
    for (r in 0 until matrix.size) {
        for (c in 0 until matrix[0].size) {
            if (matrix[r][c] < 0) {
                return false
            }
        }
    }
    return true
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/linkedListConstruction/solution1/linkedListConstruction.kt

```
package ae.medium.LinkedListConstruction.solution1

class Node(value: Int) {
    val value = value
    var prev: Node? = null
    var next: Node? = null
}

class DoublyLinkedList {
    private var head: Node? = null
    private var tail: Node? = null

    fun setHead(node: Node) {
        if (head == null) {
            head = node
            tail = node
            return
        }

        insertBefore(head!!, node)
    }

    fun setTail(node: Node) {
        if (tail == null) {
            setHead(node)
            return
        }

        insertAfter(tail!!, node)
    }

    fun insertBefore(node: Node, nodeToInsert: Node) {
        if (nodeToInsert == head && nodeToInsert == tail) {
            return
        }

        remove(nodeToInsert)
        nodeToInsert.prev = node.prev
        nodeToInsert.next = node

        if (node.prev == null) {
            head = nodeToInsert
        } else {
            node.prev!!.next = nodeToInsert
        }

        node.prev = nodeToInsert
    }

    fun insertAfter(node: Node, nodeToInsert: Node) {
        if (nodeToInsert == head && nodeToInsert == tail) {
            return
        }
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/linkedListConstruction/solution1/linkedListConstruction.kt

```
}

remove(nodeToInsert)
nodeToInsert.next = node.next
nodeToInsert.prev = node

if (node.next == null) {
    tail = nodeToInsert
} else {
    node.next!!.prev = nodeToInsert
}

node.next = nodeToInsert
}

fun insertAtPosition(position: Int, nodeToInsert: Node) {
    // Unnecessary because this logic will be handled by the next
commented line
    if (position == 1) {
        setHead(nodeToInsert)
        return
    }

    var currentNode: Node? = head
    var currentPosition = 1

    while (currentNode != null && currentPosition != position) {
        currentNode = currentNode.next
        currentPosition += 1
    }

    if (currentNode == null) {
        setTail(nodeToInsert)
    } else {
        // If currentNode is head, then this will effectively insert
the new node in the head
        insertBefore(currentNode, nodeToInsert)
    }
}

fun removeNodesWithValue(value: Int) {
    var current = head

    while (current != null) {
        val nextNode = current.next

        if (current.value == value) {
            remove(current)
        }

        current = nextNode
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/linkedListConstruction/solution1/linkedListConstruction.kt

```
}

fun remove(node: Node) {
    if (node == head) {
        head = head!!.next
    }

    if (node == tail) {
        tail = tail!!.prev
    }

    removeBindings(node)
}

fun containsNodeWithValue(value: Int): Boolean {
    var currentNode = head

    while (currentNode != null && currentNode.value != value) {
        currentNode = currentNode.next
    }

    return currentNode != null
}

fun getHead(): Node? { return this.head }

fun getTail(): Node? { return this.tail }

fun removeBindings(node: Node) {
    if (node.prev != null) {
        node.prev!!.next = node.next
    }

    if (node.next != null) {
        node.next!!.prev = node.prev
    }

    node.prev = null
    node.next = null
}
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/maxSubsetSumNoAdjacent/solution1/maxSubsetSumNoAdjacent.kt

```
package ae.medium.maxSubsetSumNoAdjacent.solution1

import kotlin.math.max

fun maxSubsetSumNoAdjacent(array: List<Int>): Int {
    if (array.isEmpty()) return 0
    if (array.size == 1) return array[0]

    val maxSums = MutableList(array.size) { 0 }
    maxSums[0] = array[0]
    maxSums[1] = max(array[0], array[1])

    for (i in 2 .. array.lastIndex) {
        maxSums[i] = max(maxSums[i - 1], maxSums[i - 2] + array[i])
    }

    return maxSums.last()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/maxSubsetSumNoAdjacent/solution2/maxSubsetSumNoAdjacent.kt

```
package ae.medium.maxSubsetSumNoAdjacent.solution2

import kotlin.math.max

fun maxSubsetSumNoAdjacent(array: List<Int>): Int {
    if (array.isEmpty()) return 0
    if (array.size == 1) return array[0]

    var first = array[0]
    var second = max(array[0], array[1])

    for (i in 2 .. array.lastIndex) {
        second = max(second, first + array[i]).also {
            first = second
        }
    }

    return second
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/suffixTrieConstruction/solution1/suffixTrieConstruction.kt

```
package ae.medium.suffixTrieConstruction.solution1

data class TrieNode(
    val children: MutableMap<Char, TrieNode> = mutableMapOf<Char, TrieNode>()
)

class SuffixTrie(str: String) {
    val endSymbol = '*'
    var root = TrieNode()

    init { populate(str) }

    fun populate(str: String) {
        for (i in 0 .. str.lastIndex) {
            insertSubstringStartingAt(i, str)
        }
    }

    fun insertSubstringStartingAt(idx: Int, str: String) {
        var currentNode = root

        for (i in idx .. str.lastIndex) {
            if (!currentNode.children.containsKey(str[i])) {
                currentNode.children[str[i]] = TrieNode()
            }

            currentNode = currentNode.children[str[i]]!!
        }

        currentNode.children[endSymbol] = TrieNode()
    }

    fun contains(str: String): Boolean {
        var currentNode = root

        for (c in str) {
            if (!currentNode.children.containsKey(c)) {
                return false
            }

            currentNode = currentNode.children[c]!!
        }

        return currentNode.children.containsKey(endSymbol)
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/youngestCommonAncestor/solution1/youngestCommonAncestor.kt

```
package ae.medium.youngestCommonAncestor.solution1

class AncestralTree(name: Char) {
    val name = name
    var ancestor: AncestralTree? = null
}

fun getYoungestCommonAncestor(topAncestor: AncestralTree, descendantOne: AncestralTree, descendantTwo: AncestralTree): AncestralTree? {
    val depthOne = getDepth(descendantOne, topAncestor)
    val depthTwo = getDepth(descendantTwo, topAncestor)
    return if (depthOne > depthTwo) {
        backtrackAncestralTree(descendantOne, descendantTwo, depthOne - depthTwo)
    } else {
        backtrackAncestralTree(descendantTwo, descendantOne, depthTwo - depthOne)
    }
}

fun getDepth(descendant: AncestralTree, topAncestor: AncestralTree): Int {
    var depth = 0
    var descendantNode = descendant

    while (descendantNode != topAncestor) {
        depth += 1
        descendantNode = descendantNode.ancestor!!
    }

    return depth
}

fun backtrackAncestralTree(lowerDescendant: AncestralTree, higherDescendant: AncestralTree?, diff: Int): AncestralTree? {
    var diffValue = diff

    var lowerDescendantNode: AncestralTree? = lowerDescendant
    var higherDescendantNode: AncestralTree? = higherDescendant

    while (diffValue > 0) {
        lowerDescendantNode = lowerDescendantNode!!.ancestor
        diffValue -= 1
    }

    while (lowerDescendantNode != higherDescendantNode) {
        lowerDescendantNode = lowerDescendantNode?.ancestor
        higherDescendantNode = higherDescendantNode?.ancestor
    }

    return lowerDescendantNode
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minMaxStackConstruction/solution1/minMaxStackConstruction.kt

```
package ae.medium.minMaxStackConstruction.solution1

import kotlin.math.min
import kotlin.math.max

open class MinMaxStack() {
    val stack = mutableListOf<Int>()
    val minMaxStack = mutableListOf<MinMax>()

    fun peek(): Int? {
        return if (stack.isNotEmpty()) stack.last() else null
    }

    fun pop(): Int? {
        return if (stack.isNotEmpty()) {
            minMaxStack.removeAt(minMaxStack.lastIndex)
            stack.removeAt(stack.lastIndex)
        } else null
    }

    fun push(number: Int) {
        val minElement = if (minMaxStack.isNotEmpty()) {
            val lastMinElement = minMaxStack.last().min
            min(lastMinElement, number)
        } else number

        val maxElement = if (minMaxStack.isNotEmpty()) {
            val lastMaxElement = minMaxStack.last().max
            max(lastMaxElement, number)
        } else number

        minMaxStack.add(MinMax(minElement, maxElement))
        stack.add(number)
    }

    fun getMin(): Int? {
        return if (stack.isNotEmpty()) {
            minMaxStack.last().min
        } else null
    }

    fun getMax(): Int? {
        return if (stack.isNotEmpty()) {
            minMaxStack.last().max
        } else null
    }
}

data class MinMax(val min: Int, val max: Int)
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/findKthLargestValueInBST/solution1/findKthLargestValueInBST.kt

```
package ae.medium.findKthLargestValueInBST.solution1
```

```
open class BST(value: Int) {  
    var value = value  
    var left: BST? = null  
    var right: BST? = null  
}  
  
fun findKthLargestValueInBst(tree: BST, k: Int): Int {  
    val treeInfo = TreeInfo(0, -1)  
    reverseInorderTraverse(tree, k, treeInfo)  
    return treeInfo.latestVisitedNodeValue  
}  
  
data class TreeInfo(var numberofNodesVisited: Int, var latestVisitedNodeValue: Int)  
  
fun reverseInorderTraverse(node: BST?, k: Int, treeInfo: TreeInfo) {  
    if (node == null || treeInfo.numberofNodesVisited >= k) return  
  
    reverseInorderTraverse(node.right, k, treeInfo)  
  
    if (treeInfo.numberofNodesVisited < k) {  
        treeInfo.numberofNodesVisited += 1  
        treeInfo.latestVisitedNodeValue = node.value  
  
        reverseInorderTraverse(node.left, k, treeInfo)  
    }  
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/heightBalancedBinaryTree/solution1/heightBalancedBinaryTree.kt

```
package ae.medium.heightBalancedBinaryTree.solution1

import kotlin.math.abs
import kotlin.math.max

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun heightBalancedBinaryTree(tree: BinaryTree): Boolean {
    return getTreeInfo(tree).isBalanced
}

data class TreeInfo(var isBalanced: Boolean, var height: Int)

fun getTreeInfo(node: BinaryTree?): TreeInfo {
    node ?: return TreeInfo(true, -1)

    val leftSubtreeInfo = getTreeInfo(node.left)
    val rightSubtreeInfo = getTreeInfo(node.right)

    val isBalanced = leftSubtreeInfo.isBalanced && rightSubtreeInfo.
        isBalanced &&
        abs(leftSubtreeInfo.height - rightSubtreeInfo.height) <= 1
    val height = max(leftSubtreeInfo.height, rightSubtreeInfo.height) + 1
    return TreeInfo(isBalanced, height)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/numberOfWaysToMakeChange/solution1/numberOfWaysToMakeChange.kt

```
package ae.medium.numberOfWaysToMakeChange.solution1

fun numberOfWaysToMakeChange(n: Int, denoms: List<Int>): Int {
    val ways = MutableList(n + 1) { 0 }
    ways[0] = 1

    denoms.forEach { denom ->
        for (amount in denom .. n) {
            ways[amount] += ways[amount - denom]
        }
    }

    return ways[n]
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/mergeOverlappingIntervals/solution1/mergeOverlappingIntervals.kt

```
package ae.medium.mergeOverlappingIntervals.solution1

import kotlin.math.max

fun mergeOverlappingIntervals(intervals: List<List<Int>>): List<List<Int>>
>> {

    // Sort intervals by start time using sortedWith
    // And map to mutable data structures
    val sortedIntervals = intervals.sortedWith(Comparator<List<Int>> { i1,
i2 ->
        i1[0].compareTo(i2[0])
    }).map { it.toMutableList() }

    val mergedIntervals = mutableListOf<MutableList<Int>>()
    // Push first interval in merged intervals
    mergedIntervals.add(sortedIntervals[0])

    // Iterate through all of the remaining intervals
    for (i in 1 .. intervals.lastIndex) {
        // Get the next interval
        val nextInterval = sortedIntervals[i]

        // Get last element in last merged intervals
        val currentInterval = mergedIntervals.last()

        // If next interval starts after the current interval finishes
        if (nextInterval[0] > currentInterval[1]) {
            // Push the new interval into merged intervals as is. This is
            // non overlapping
            mergedIntervals.add(nextInterval)
        } else {
            // Current interval and next interval is overlapping. So this
            // will not be inserted as a new interval.
            // Current interval's end time will be the end time of the
            // interval which finishes later
            currentInterval[1] = max(currentInterval[1], nextInterval[1])
        }
    }

    return mergedIntervals.map { it.toList() }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minNumberOfCoinsForChange/solution1/minNumberOfCoinsForChange.kt

```
package ae.medium.minNumberOfCoinsForChange.solution1

import kotlin.math.min

fun minNumberOfCoinsForChange(n: Int, denoms: List<Int>): Int {
    val numCoins = MutableList(n + 1) { Integer.MAX_VALUE }
    numCoins[0] = 0

    denoms.forEach { denom ->
        for (amount in denom .. n) {
            val toCompare = if (numCoins[amount - denom] == Integer.
MAX_VALUE) {
                numCoins[amount - denom]
            } else {
                1 + numCoins[amount - denom]
            }

            numCoins[amount] = min(numCoins[amount], toCompare)
        }
    }

    return if (numCoins[n] == Integer.MAX_VALUE) -1 else numCoins[n]
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/minimumCharactersForWords/solution1/minimumCharactersForWords.kt

```
package ae.medium.minimumCharactersForWords.solution1

import kotlin.math.max

fun minimumCharactersForWords(words: List<String>): List<Char> {
    val maxFrequencies = mutableMapOf<Char, Int>()

    for (word in words) {
        val charFrequencies = computeCharFrequencies(word)
        updateMaxFrequencies(charFrequencies, maxFrequencies)
    }
    return makeListFromMaxFrequencies(maxFrequencies)
}

fun computeCharFrequencies(string: String): Map<Char, Int> {
    val charFreq = mutableMapOf<Char, Int>()

    for (c in string) {
        if (!charFreq.containsKey(c)) {
            charFreq[c] = 0
        }

        charFreq[c] = charFreq[c]!! + 1
    }

    return charFreq
}

fun updateMaxFrequencies(charFrequencies: Map<Char, Int>, maxFrequencies: MutableMap<Char, Int>) {
    charFrequencies.forEach { (char, freq) ->
        if (!maxFrequencies.containsKey(char)) {
            maxFrequencies[char] = freq
        } else {
            maxFrequencies[char] = max(maxFrequencies[char]!!, freq)
        }
    }
}

fun makeListFromMaxFrequencies(frequencies: Map<Char, Int>): List<Char> {
    val charList = mutableListOf<Char>()

    frequencies.forEach { (char, freq) ->
        charList.addAll(List(freq) { char })
    }

    return charList
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/longestPalindromicSubstring/solution1/longestPalindromicSubstring.kt

```
package ae.medium.longestPalindromicSubstring.solution1

fun longestPalindromicSubstring(string: String): String {
    var currentLongest = listOf(0, 1)

    for (i in 1 .. string.lastIndex) {
        val odd = getLongestPalindromeFrom(string, i - 1, i + 1)
        val even = getLongestPalindromeFrom(string, i - 1, i)

        val longest = if (odd[1] - odd[0] > even[1] - even[0]) odd else
even
            currentLongest = if (longest[1] - longest[0] > currentLongest[1] - currentLongest[0]) longest else currentLongest
    }

    return string.substring(currentLongest[0], currentLongest[1])
}

fun getLongestPalindromeFrom(string: String, leftIdx: Int, rightIdx: Int): List<Int> {
    var currentLeftIdx = leftIdx
    var currentRightIdx = rightIdx

    while (currentLeftIdx >= 0 && currentRightIdx < string.length) {
        if (string[currentLeftIdx] != string[currentRightIdx]) {
            break
        }

        currentLeftIdx -= 1
        currentRightIdx += 1
    }

    // At this point leftIdx and rightIdx are not part of palindrome.
    // Or they are out of bounds.
    // So the returned left index is inclusive but returned right index is
exclusive.
    return listOf(currentLeftIdx + 1, currentRightIdx)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/numberOfWaysToTraverseGraph/solution1/numberOfWaysToTraverseGraph.kt

```
package ae.medium.numberOfWaysToTraverseGraph.solution1

fun numberOfWaysToTraverseGraph(width: Int, height: Int): Int {
    val memo = MutableList(height) { MutableList(width) { -1 } }
    return recurse(0, 0, height - 1, width - 1, memo)
}

fun recurse(r: Int, c: Int, bottom: Int, right: Int, memo: MutableList<MutableList<Int>>): Int {
    if (r == bottom || c == right) return 1

    if (memo[r][c] != -1) return memo[r][c]

    // Go down + go right
    memo[r][c] = recurse(r + 1, c, bottom, right, memo) + recurse(r, c + 1,
    , bottom, right, memo)
    return memo[r][c]
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/numberOfWaysToTraverseGraph/solution2/numberOfWaysToTraverseGraph.kt

```
package ae.medium.numberOfWaysToTraverseGraph.solution2

fun numberOfWaysToTraverseGraph(width: Int, height: Int): Int {
    val memo = MutableList(height + 1) { MutableList(width + 1) { -1 } }
    return recurse(width, height, memo)
}

fun recurse(width: Int, height: Int, memo: MutableList<MutableList<Int>>): Int {
    if (width == 1 || height == 1) return 1

    if (memo[height][width] != -1) return memo[height][width]

    memo[height][width] = recurse(width - 1, height, memo) + recurse(width,
        height - 1, memo)

    return memo[height][width]
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/medium/numberOfWaysToTraverseGraph/solution3/numberOfWaysToTraverseGraph.kt

```
package ae.medium.numberOfWaysToTraverseGraph.solution3

fun numberOfWaysToTraverseGraph(width: Int, height: Int): Int {
    val dp = mutableListOf(height) { mutableListOf(width) { 0 } }

    for (r in 0 until height) {
        for (c in 0 until width) {
            if (r == 0 || c == 0) {
                dp[r][c] = 1
            } else {
                dp[r][c] = dp[r - 1][c] + dp[r][c - 1]
            }
        }
    }

    return dp[height - 1][width - 1]
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/lruCache/solution1/lruCache.kt

```
package ae.veryHard.lruCache.solution1

class LRUCache(var maxSize: Int) {

    var currentSize = 0
    val listOfMostRecent = DoublyLinkedList()
    val cache = mutableMapOf<String, DoublyLinkedListNode>()

    init {
        maxSize = if (maxSize > 1) maxSize else 1
        currentSize = 0
    }

    fun insertKeyValuePair(key: String, value: Int) {
        if (!cache.containsKey(key)) {
            if (currentSize == maxSize) {
                evictLeastRecentNode()
            } else {
                currentSize += 1
            }
            cache[key] = DoublyLinkedListNode(key, value)
        } else {
            updateValueForKey(key, value)
        }
        updateMostRecentNode(cache[key]!!)
    }

    fun getValueFromKey(key: String): Int? {
        if (!cache.containsKey(key)) {
            return null
        }
        updateMostRecentNode(cache[key]!!)
        return cache[key]!!.value
    }

    fun getMostRecentKey(): String? {
        if (listOfMostRecent.head == null) {
            return null
        }
        return listOfMostRecent.head!!.key
    }

    private fun evictLeastRecentNode() {
        val keyToRemove = listOfMostRecent.tail?.key
        keyToRemove ?: return
        listOfMostRecent.removeTail()
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/lruCache/solution1/lruCache.kt

```
    cache.remove(keyToRemove)
}

private fun updateMostRecentNode(node: DoublyLinkedListNode) {
    listOfMostRecent.setHeadTo(node)
}

private fun updateValueForKey(key: String, value: Int) {
    cache[key]!!.value = value
}
}

data class DoublyLinkedListNode(val key: String, var value: Int) {
    var prev: DoublyLinkedListNode? = null
    var next: DoublyLinkedListNode? = null

    fun removeBindings() {
        if (prev != null) {
            prev!!.next = next
        }

        if (next != null) {
            next!!.prev = prev
        }

        prev = null
        next = null
    }
}

class DoublyLinkedList {

    var head: DoublyLinkedListNode? = null
    var tail: DoublyLinkedListNode? = null

    fun setHeadTo(node: DoublyLinkedListNode) {
        if (node == head) {
            return
        }

        if (head == null) {
            head = node
            tail = node
        } else if (head == tail) {
            head = node
            head!!.next = tail
            tail!!.prev = head
        } else {
            if (node == tail) {
                removeTail()
            }
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/lruCache/solution1/lruCache.kt

```
        node.removeBindings()
        node.next = head
        head!!.prev = node
        head = node
    }
}

fun removeTail() {
    if (tail == null) {
        return
    }

    if (head == tail) {
        head = null
        tail = null
        return
    }

    tail = tail!!.prev
    tail!!.next = null
}
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/nodeSwap/solution1/nodeSwap.kt

```
package ae.veryHard.nodeSwap.solution1

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun nodeSwap(head: LinkedList?): LinkedList {
    return recurseHelper(head)!!
}

fun recurseHelper(head: LinkedList?): LinkedList? {
    if (head?.next == null) {
        return head
    }

    val firstNode = head
    val secondNode = head.next

    firstNode.next = recurseHelper(secondNode?.next)
    secondNode?.next = firstNode

    return secondNode
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/nodeSwap/solution2/nodeSwap.kt

```
package ae.veryHard.nodeSwap.solution2

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun nodeSwap(head: LinkedList): LinkedList {
    val preHead = LinkedList(-1)
    preHead.next = head

    var prev: LinkedList? = preHead

    while (prev?.next?.next != null) {
        val firstNode = prev.next!!
        val secondNode = prev.next?.next!!

        firstNode.next = secondNode.next
        secondNode.next = firstNode
        prev.next = secondNode

        prev = firstNode
    }

    return preHead.next!!
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/mergeSort/solution1/mergeSort.kt

```
package ae.veryHard.mergeSort.solution1

fun mergeSort(array: MutableList<Int>): List<Int> {
    if (array.size == 1) return array

    val middle = array.size / 2
    val leftHalf = array.subList(0, middle)
    val rightHalf = array.subList(middle, array.size)
    return mergeSortedArrays(mergeSort(leftHalf), mergeSort(rightHalf))
}

fun mergeSortedArrays(leftHalf: List<Int>, rightHalf: List<Int>): List<Int> {
    var i = 0
    var j = 0

    val resultArray = mutableListOf<Int>()

    while (i < leftHalf.size && j < rightHalf.size) {
        if (leftHalf[i] < rightHalf[j]) {
            resultArray.add(leftHalf[i])
            i += 1
        } else {
            resultArray.add(rightHalf[j])
            j += 1
        }
    }

    while (i < leftHalf.size) {
        resultArray.add(leftHalf[i])
        i += 1
    }

    while (j < rightHalf.size) {
        resultArray.add(rightHalf[j])
        j += 1
    }
}

return resultArray
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/mergeSort/solution2/mergeSort.kt

```
package ae.veryHard.mergeSort.solution2

fun mergeSort(array: MutableList<Int>): List<Int> {
    if (array.isEmpty()) return array
    // Copy the array to an auxiliary array, not to overwrite the original
    // array in the sort process
    val auxiliaryArray = array.toMutableList()
    mergeSortHelper(array, 0, array.lastIndex, auxiliaryArray)
    return array
}

fun mergeSortHelper(mainArray: MutableList<Int>, startIdx: Int, endIdx: Int,
    auxiliaryArray: MutableList<Int>) {
    if (startIdx == endIdx) return

    val middleIdx = (startIdx + endIdx) / 2

    mergeSortHelper(auxiliaryArray, startIdx, middleIdx, mainArray)
    mergeSortHelper(auxiliaryArray, middleIdx + 1, endIdx, mainArray)

    mergeSortedArrays(mainArray, startIdx, middleIdx, endIdx,
        auxiliaryArray)
}

fun mergeSortedArrays(mainArray: MutableList<Int>, startIdx: Int, middleIdx
    : Int, endIdx: Int, auxiliaryArray: MutableList<Int>) {
    var i = startIdx
    var j = middleIdx + 1
    var k = startIdx

    while (i <= middleIdx && j <= endIdx) {
        if (auxiliaryArray[i] < auxiliaryArray[j]) {
            mainArray[k] = auxiliaryArray[i]
            i += 1
            k += 1
        } else {
            mainArray[k] = auxiliaryArray[j]
            j += 1
            k += 1
        }
    }

    while (i <= middleIdx) {
        mainArray[k] = auxiliaryArray[i]
        i += 1
        k += 1
    }

    while (j <= endIdx) {
        mainArray[k] = auxiliaryArray[j]
        j += 1
        k += 1
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/mergeSort/solution2/mergeSort.kt

```
}\n}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/zipLinkedList/solution1/zipLinkedList.kt

```
package ae.veryHard.zipLinkedList.solution1

// This is an input class. Do not edit.
open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun zipLinkedList(linkedList: LinkedList): LinkedList {
    // If the linked list contains less than 3 elements
    // then there are no point to zip.
    if (linkedList.next?.next == null) {
        return linkedList
    }

    val firstHalfHead = linkedList
    val secondHalfHead = splitLinkedList(linkedList)
    val secondHalfHeadReversed = reverseLinkedList(secondHalfHead)

    return interweaveLinkedList(firstHalfHead, secondHalfHeadReversed)!!
}

fun splitLinkedList(head: LinkedList): LinkedList? {
    var slow: LinkedList? = head
    var fast: LinkedList? = head

    while (fast?.next?.next != null) {
        slow = slow?.next
        fast = fast.next?.next
    }

    val secondHalfHead = slow!!.next
    slow.next = null

    return secondHalfHead
}

fun reverseLinkedList(head: LinkedList?): LinkedList? {
    var prev: LinkedList? = null
    var current: LinkedList? = head

    while (current != null) {
        val next = current.next
        current.next = prev
        prev = current
        current = next
    }

    return prev
}

fun interweaveLinkedList(ll1: LinkedList?, ll2: LinkedList?): LinkedList? {
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/zipLinkedList/solution1/zipLinkedList.kt

```
val ll1Head = ll1

var ll1Current = ll1
var ll2Current = ll2

while (ll2Current != null) {
    val ll1Next = ll1Current?.next
    val ll2Next = ll2Current.next

    ll1Current?.next = ll2Current
    ll2Current.next = ll1Next

    ll1Current = ll1Next
    ll2Current = ll2Next
}

return ll1Head
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/aStarAlgorithm/solution1/aStarAlgorithm.kt

```
package ae.veryHard.aStarAlgorithm.solution1

fun aStarAlgorithm(startRow: Int, startCol: Int, endRow: Int, endCol: Int,
graph: List<List<Int>>): List<List<Int>> {
    // G, H and F Score
    // G Score = Distance from start node
    // H Score = Estimated distance from end node (Manhattan distance)
    // F Score = G Score (Distance from start node) + H Score (Estimated
    // distance from end node)

    // Build graph
    val nodeGraph = buildGraph(graph)

    // Get start and end node
    val startNode = nodeGraph[startRow][startCol]
    val endNode = nodeGraph[endRow][endCol]

    // Start node's G Score is 0
    startNode.distanceFromStart = 0

    // Calculate H Score for start node
    startNode.estimatedDistanceToEnd = calculateManhattanDistance(startNode,
        endNode)

    // We need a heap because at each step we need to find the node using
    // which the start node will be closest to the destination
    val nodesToVisit = MinHeap()
    // Insert the starting node to the heap
    nodesToVisit.insert(startNode)

    // Until the heap is empty
    while (!nodesToVisit.isEmpty()) {
        // Remove the root node from the heap which **using which the start
        // node will be closest to the end node**
        val currentNode = nodesToVisit.remove()

        // If the node popped off is the end node then break. We have
        // reached the destination
        if (currentNode == endNode)
            break

        // Get all of the neighboring nodes of the current node
        val neighbors = getNeighbors(currentNode, nodeGraph)

        // Distance to neighbor node is the distance of the current node
        // from the start node + 1
        val distanceToNeighbor = currentNode.distanceFromStart + 1

        for (neighbor in neighbors) {

            // If neighbor value is 1 then this is an obstacle. This can't
            // be passed through, so continue.
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/aStarAlgorithm/solution1/aStarAlgorithm.kt

```
    if (neighbor.value == 1)
        continue

        // The the newly calculated distance is greater than or equal
        to the already existing distance then it's of no use, discard it
        if (distanceToNeighbor >= neighbor.distanceFromStart)
            continue

        // Set neighbor's distance from start is equal to the distance
        to neighbor
        neighbor.distanceFromStart = distanceToNeighbor

        // Using this neighbor the estimated distance from start node
        to destination node is *** distance to neighbor + manhattan distance from
        this node to destination ***
        neighbor.estimatedDistanceToEnd = distanceToNeighbor +
calculateManhattanDistance(neighbor, endNode)

        // Neighbor came from the current node so mark it with that.
        Needed for build the sequence to the start node.
        neighbor.cameFrom = currentNode

        // If the heap contains this node then update it
        if (nodesToVisit.contains(neighbor)) {
            nodesToVisit.update(neighbor)
        } else {

            // Otherwise insert it
            nodesToVisit.insert(neighbor)
        }
    }

    return buildResult(endNode, nodeGraph)
}

fun buildGraph(nodes: List<List<Int>>): List<List<Node>> {

    val rows = nodes.size
    val cols = nodes[0].size

    val graph = MutableList(nodes.size) { mutableListOf<Node>() }

    for (r in 0 until rows) {
        for (c in 0 until cols) {
            graph[r].add(Node(r, c, nodes[r][c]))
        }
    }

    return graph.map { it.toList() }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/aStarAlgorithm/solution1/aStarAlgorithm.kt

```
fun calculateManhattanDistance(a: Node, b: Node): Int {
    return (a.row - b.row) + (a.col - b.col)
}

fun getNeighbors(current: Node, graph: List<List<Node>>): List<Node> {
    val numRows = graph.size
    val numCols = graph[0].size

    val currentRow = current.row
    val currentCol = current.col

    val neighbors = mutableListOf<Node>()

    if (currentRow - 1 >= 0) {
        neighbors.add(graph[currentRow - 1][currentCol])
    }

    if (currentCol - 1 >= 0) {
        neighbors.add(graph[currentRow][currentCol - 1])
    }

    if (currentRow + 1 < numRows) {
        neighbors.add(graph[currentRow + 1][currentCol])
    }

    if (currentCol + 1 < numCols) {
        neighbors.add(graph[currentRow][currentCol + 1])
    }

    return neighbors
}

fun buildResult(endNode: Node, graph: List<List<Node>>): List<List<Int>> {
    // If end node doesn't contain any node which it came from then end node is not reachable.
    if (endNode.cameFrom == null) {
        return listOf()
    }

    // Backtrack through the result starting from the end node
    val result = mutableListOf<List<Int>>()
    var currentNode: Node? = endNode

    while (currentNode != null) {
        result.add(listOf(currentNode.row, currentNode.col))
        currentNode = currentNode.cameFrom
    }

    return result.reversed()
}

data class Node(val row: Int, val col: Int, val value: Int) {
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/aStarAlgorithm/solution1/aStarAlgorithm.kt

```
val id = "$row-$col"

// Distance from start node - G Score
var distanceFromStart = Integer.MAX_VALUE

// Estimated (or Manhattan) distance to end node - H Score
var estimatedDistanceToEnd = Integer.MAX_VALUE

// F Score = G Score + H Score

var cameFrom: Node? = null
}

// This heap's sorting parameter is estimated distance to end (H Score)
class MinHeap {

    val heap = mutableListOf<Node>()

    // This is a map between node's id and it's index in the underlying
    // heap array
    val nodePositionsToHeap = mutableMapOf<String, Int>()

    fun siftDown(currentIdx: Int, endIdx: Int, heap: List<Node>) {

        var mutableCurrentIdx = currentIdx

        var childOneIdx = mutableCurrentIdx * 2 + 1

        while (childOneIdx <= endIdx) {
            val childTwoIdx = if (mutableCurrentIdx * 2 + 2 <= endIdx)
                mutableCurrentIdx * 2 + 2 else -1
            var idxToSwap = if (childTwoIdx != -1 && heap[childTwoIdx].estimatedDistanceToEnd < heap[childOneIdx].estimatedDistanceToEnd) {
                childTwoIdx
            } else childOneIdx

            if (heap[idxToSwap].estimatedDistanceToEnd < heap[mutableCurrentIdx].estimatedDistanceToEnd) {
                swap(mutableCurrentIdx, idxToSwap)
                mutableCurrentIdx = idxToSwap
                childOneIdx = mutableCurrentIdx * 2 + 1
            } else {
                // Already in the correct position
                return
            }
        }
    }

    fun siftUp(currentIdx: Int) {
        var mutableCurrentIdx = currentIdx

        var parentIdx = (mutableCurrentIdx - 1) / 2
```

```

        while (mutableCurrentIdx > 0 && heap[mutableCurrentIdx].
estimatedDistanceToEnd < heap[parentIdx].estimatedDistanceToEnd) {
            swap(mutableCurrentIdx, parentIdx)
            mutableCurrentIdx = parentIdx
            parentIdx = (mutableCurrentIdx - 1) / 2
        }
    }

fun remove(): Node {
    swap(heap.lastIndex, 0)
    val nodeToRemove = heap.removeAt(heap.lastIndex)
    siftDown(0, heap.lastIndex, heap)
    return nodeToRemove
}

fun insert(node: Node) {
    heap.add(node)
    nodePositionsToHeap[node.id] = heap.lastIndex
    siftUp(heap.lastIndex)
}

fun isEmpty(): Boolean = heap.size == 0

fun contains(node: Node): Boolean {
    // We can search if the item exists in constant time by searching
    for it in the hash map.
    return nodePositionsToHeap.containsKey(node.id)
}

fun update(node: Node) {
    // The node object is already updated.
    // So here update means that we need to sift it up to the
    appropriate position.
    siftUp(nodePositionsToHeap[node.id]!!)
}

fun swap(i: Int, j: Int) {
    val temp = heap[i]
    heap[i] = heap[j]
    heap[j] = temp

    // The items are swapped in the array. i went to j and j went to i.
    // So now in the map we need to update their index.
    // i and j are swapped before. So now assign i's value to i and j's
    value to j.
    nodePositionsToHeap[heap[i].id] = i
    nodePositionsToHeap[heap[j].id] = j
}
}

```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/rectangleMania/solution1/rectangleMania.kt

```
package ae.veryHard.rectangleMania.solution1

fun rectangleMania(coords: List<List<Int>>): Int {
    val coordsSet = getCoordsSet(coords)
    return getRectangleCount(coords, coordsSet)
}

fun getCoordsSet(coords: List<List<Int>>): Set<String> {
    return coords.map {
        coordsToString(it)
    }.toSet()
}

fun coordsToString(coord: List<Int>) = "${coord[0]}|${coord[1]}"

fun getRectangleCount(coords: List<List<Int>>, coordsSet: Set<String>): Int {
    var rectangleCount = 0

    for (point1 in coords) {
        for (point2 in coords) {

            if (!isOnUpperRightDiagonal(point1, point2)) {
                continue
            }

            val (x1, y1) = point1
            val (x2, y2) = point2

            val topLeftPointKey = coordsToString(listOf(x1, y2))
            val bottomRightPointKey = coordsToString(listOf(x2, y1))

            if (coordsSet.contains(topLeftPointKey) && coordsSet.contains(
                bottomRightPointKey)) {
                rectangleCount++
            }
        }
    }

    return rectangleCount
}

fun isOnUpperRightDiagonal(point1: List<Int>, point2: List<Int>): Boolean {
    val (x1, y1) = point1
    val (x2, y2) = point2

    return x2 > x1 && y2 > y1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/rectangleMania/solution2/rectangleMania.kt

```
package ae.veryHard.rectangleMania.solution2

const val UP = "up"
const val RIGHT = "right"
const val DOWN = "down"

fun rectangleMania(coords: List<List<Int>>): Int {
    val coordsMap = getCoordsMap(coords)
    return getRectangleCount(coords, coordsMap)
}

fun getCoordsMap(coords: List<List<Int>>): Map<Char, Map<Int, List<List<Int>>>> {
    val coordsMap = mutableMapOf(
        'x' to mutableMapOf<Int, MutableList<List<Int>>>(),
        'y' to mutableMapOf<Int, MutableList<List<Int>>>()
    )

    for (coord in coords) {
        val (x, y) = coord

        if (!coordsMap['x']!![x].containsKey(x)) {
            coordsMap['x']!![x] = mutableListOf()
        }
        coordsMap['x']!![x]!![x]!!.add(coord)

        if (!coordsMap['y']!![y].containsKey(y)) {
            coordsMap['y']!![y] = mutableListOf()
        }
        coordsMap['y']!![y]!![y]!!.add(coord)
    }

    return coordsMap.mapValues { (_, axesMap) ->
        axesMap.mapValues { (_, coordList) -> coordList.toList() }
    }
}

fun getRectangleCount(coords: List<List<Int>>, coordsMap: Map<Char, Map<Int, List<List<Int>>>>): Int {
    var rectangleCount = 0

    for (coord in coords) {
        rectangleCount += clockwiseCountRectangle(coords, coordsMap, coord,
        UP, coord[1])
    }

    return rectangleCount
}

fun clockwiseCountRectangle(coords: List<List<Int>>, coordsMap: Map<Char, Map<Int, List<List<Int>>>>, coord: List<Int>, direction: String, lowerLeftY
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/rectangleMania/solution2/rectangleMania.kt

```
: Int): Int {
    val (x, y) = coord

    return if (direction == DOWN) {
        for ((nextX, nextY) in coordsMap['x']!![x]!!) {
            if (nextY == lowerLeftY) {
                return 1
            }
        }
        0
    } else {
        var rectangleCount = 0

        if (direction == UP) {
            for (nextCoord in coordsMap['x']!![x]!!) {
                if (nextCoord[1] > y) {
                    rectangleCount += clockwiseCountRectangle(coords,
coordsMap, nextCoord, RIGHT, lowerLeftY)
                }
            }
        } else if (direction == RIGHT) {
            for (nextCoord in coordsMap['y']!![y]!!) {
                if (nextCoord[0] > x) {
                    rectangleCount += clockwiseCountRectangle(coords,
coordsMap, nextCoord, DOWN, lowerLeftY)
                }
            }
        }

        rectangleCount
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/squareOfZeroes/solution1/squareOfZeroes.kt

```
package ae.veryHard.squareOfZeroes.solution1

fun squareOfZeroes(matrix: List<List<Int>>): Boolean {

    val len = matrix.size

    // For each element
    for (r in 0 until len) {
        for (c in 0 until len) {

            // Consider this cell as the top left corner of a square
            // Start with square of side length 2
            var sideLen = 2

            // While side of length sideLen from this cell are in bound
            while (r + sideLen - 1 < len && c + sideLen - 1 < len) {

                // Call function to check if this square is a square of
                zero
                if (isSquareOfZeroes(matrix, r, c, r + sideLen - 1, c +
                sideLen - 1)) {
                    return true
                }

                // Try to increase sideLen from 2 to 3, 4, 5 etc.
                sideLen++
            }
        }
    }

    return false
}

fun isSquareOfZeroes(matrix: List<List<Int>>, r1: Int, c1: Int, r2: Int, c2
: Int): Boolean {
    // For all rows check if start and end columns are not 0
    // If not then square is not square of zeroes
    for (r in r1 .. r2) {
        if (matrix[r][c1] != 0 || matrix[r][c2] != 0) {
            return false
        }
    }

    // For all cols check if start and end rows are not 0
    // If not then square is not square of zeroes
    for (c in c1 .. c2) {
        if (matrix[r1][c] != 0 || matrix[r2][c] != 0) {
            return false
        }
    }

    return true
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/squareOfZeroes/solution1/squareOfZeroes.kt
}

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/squareOfZeroes/solution2/squareOfZeroes.kt

```
package ae.veryHard.squareOfZeroes.solution2

// O(n^3) Time | O(n ^ 2) Space
fun squareOfZeroes(matrix: List<List<Int>>): Boolean {
    val infoMatrix = computeInfoMatrix(matrix)

    val len = matrix.size

    // For each element
    for (r in 0 until len) {
        for (c in 0 until len) {

            // Consider this cell as the top left corner of a square
            // Start with square of side length 2
            var sideLen = 2

            // While side of length sideLen from this cell are in bound
            while (r + sideLen - 1 < len && c + sideLen - 1 < len) {

                // Call function to check if this square is a square of
                zero
                if (isSquareOfZeroes(infoMatrix, r, c, r + sideLen - 1, c
+ sideLen - 1)) {
                    return true
                }

                // Try to increase sideLen from 2 to 3, 4, 5 etc.
                sideLen++
            }
        }
    }

    return false
}

data class CellInfo(var rightZeroesCount: Int, var bottomZeroesCount: Int)

fun computeInfoMatrix(matrix: List<List<Int>>): List<List<CellInfo>> {
    val len = matrix.size

    // Info matrix of n * n
    val infoMatrix = mutableListOf(len) {
        mutableListOf(len) { CellInfo(0, 0) }
    }

    for (r in 0 until len) {
        for (c in 0 until len) {
            // Mark all of the cells which has '0' value with {1, 1}
            // So the assumption is that in this position
            // there are 1 consecutive 0 to the right and 1 consecutive 0
            to bottom
            // So only this '0' cell
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/squareOfZeroes/solution2/squareOfZeroes.kt

```
    if (matrix[r][c] == 0) {
        infoMatrix[r][c] = CellInfo(1, 1)
    }
}

// Start from last row last column and decrement towards top-left
// For all rows starting from last to the 1st row
for (r in len - 1 downTo 0) {
    // For all columns starting from last to the 1st column
    for (c in len - 1 downTo 0) {

        // If value is 1, then there are 0 consecutive 0 value in
        bottom and right
        // So continue
        if (matrix[r][c] == 1) {
            continue
        }

        // After last row there is no row at it's bottom
        if (r < len - 1) {
            infoMatrix[r][c].bottomZeroesCount += infoMatrix[r + 1][c].
bottomZeroesCount
        }

        // After last column there is no column at it's right
        if (c < len - 1) {
            infoMatrix[r][c].rightZeroesCount += infoMatrix[r][c + 1].
rightZeroesCount
        }
    }
}

// Return the accumulated info matrix
return infoMatrix
}

fun isSquareOfZeroes(infoMatrix: List<List<CellInfo>>, r1: Int, c1: Int, r2
: Int, c2: Int): Boolean {
    // Calculate width and height
    val width = c2 - c1 + 1
    val height = r2 - r1 + 1

    // If the top left point has more than or equal to amount
    // of consecutive zeroes in the bottom direction
    val isLeftColZeroes = infoMatrix[r1][c1].bottomZeroesCount >= height

    // If the top left point has more than or equal to amount
    // of consecutive zeroes in the right direction
    val isTopRowZeroes = infoMatrix[r1][c1].rightZeroesCount >= width

    // If the top right point has more than or equal to amount
```

```
File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/squareOfZeroes/solution2/squareOfZeroes.kt

// of consecutive zeroes in the bottom direction
val isRightColZeroes = infoMatrix[r1][c2].bottomZeroesCount >= height

// If the bottom left point has more than or equal to amount
// of consecutive zeroes in the right direction
val isBottomRowZeroes = infoMatrix[r2][c1].rightZeroesCount >= width

// If all of these are true then we have a square of zeroes.
return isLeftColZeroes && isTopRowZeroes && isRightColZeroes &&
isBottomRowZeroes
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/squareOfZeroes/solution3/squareOfZeroes.kt

```
package ae.veryHard.squareOfZeroes.solution3

// O(n^3) Time | O(n ^ 3) Space
fun squareOfZeroes(matrix: List<List<Int>>): Boolean {
    val infoMatrix = computeInfoMatrix(matrix)

    val len = matrix.size

    // Eventually we will consider same square multiple time.
    // So we need to cache the result of squares in a cache.
    val cache = mutableMapOf<String, Boolean>()

    // Start from the whole square to check if it is a square of zeroes.
    return containsSquareOfZeroes(infoMatrix, 0, 0, len - 1, len - 1, cache)
}

fun containsSquareOfZeroes(infoMatrix: List<List<CellInfo>>, r1: Int, c1: Int, r2: Int, c2: Int, cache: MutableMap<String, Boolean>): Boolean {
    // If r1 passed r2 or c1 passed c2
    if (r2 <= r1 || c2 <= c1) return false

    // Compute the key
    val cellKey = "$r1-$c1-$r2-$c2"

    // If we found the key in the cache, then return it from cache.
    if (cache.containsKey(cellKey)) {
        return cache[cellKey]!!
    }

    // If current square is square of zeroes
    cache[cellKey] = isSquareOfZeroes(infoMatrix, r1, c1, r2, c2) ||
        // Shrink top left corner inward
        containsSquareOfZeroes(infoMatrix, r1 + 1, c1 + 1, r2, c2,
        cache) ||
        // Shrink top right corner inward
        containsSquareOfZeroes(infoMatrix, r1 + 1, c1, r2, c2 - 1,
        cache) ||
        // Shrink bottom left corner inward
        containsSquareOfZeroes(infoMatrix, r1, c1 + 1, r2 - 1, c2,
        cache) ||
        // Shrink bottom right corner inward
        containsSquareOfZeroes(infoMatrix, r1, c1, r2 - 1, c2 - 1,
        cache) ||
        // Shrink all four corners together inward
        containsSquareOfZeroes(infoMatrix, r1 + 1, c1 + 1, r2 - 1, c2
        - 1, cache)

    return cache[cellKey]!!
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/squareOfZeroes/solution3/squareOfZeroes.kt

```
data class CellInfo(var rightZeroesCount: Int, var bottomZeroesCount: Int)

fun computeInfoMatrix(matrix: List<List<Int>>): List<List<CellInfo>> {
    val len = matrix.size

    // Info matrix of n * n
    val infoMatrix = mutableListOf(len) {
        MutableList(len) { CellInfo(0, 0) }
    }

    for (r in 0 until len) {
        for (c in 0 until len) {
            // Mark all of the cells which has '0' value with {1, 1}
            // So the assumption is that in this position
            // there are 1 consecutive 0 to the right and 1 consecutive 0
            // to bottom
            // So only this '0' cell
            if (matrix[r][c] == 0) {
                infoMatrix[r][c] = CellInfo(1, 1)
            }
        }
    }

    // Start from last row last column and decrement towards top-left
    // For all rows starting from last to the 1st row
    for (r in len - 1 downTo 0) {
        // For all columns starting from last to the 1st column
        for (c in len - 1 downTo 0) {

            // If value is 1, then there are 0 consecutive 0 value in
            // bottom and right
            // So continue
            if (matrix[r][c] == 1) {
                continue
            }

            // After last row there is no row at it's bottom
            if (r < len - 1) {
                infoMatrix[r][c].bottomZeroesCount += infoMatrix[r + 1][c].
                bottomZeroesCount
            }

            // After last column there is no column at it's right
            if (c < len - 1) {
                infoMatrix[r][c].rightZeroesCount += infoMatrix[r][c + 1].
                rightZeroesCount
            }
        }
    }

    // Return the accumulated info matrix
    return infoMatrix
}
```

```
File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/squareOfZeroes/solution3/squareOfZeroes.kt
}

fun isSquareOfZeroes(infoMatrix: List<List<CellInfo>>, r1: Int, c1: Int, r2: Int, c2: Int): Boolean {
    // Calculate width and height
    val width = c2 - c1 + 1
    val height = r2 - r1 + 1

    // If the top left point has more than or equal to amount
    // of consecutive zeroes in the bottom direction
    val isLeftColZeroes = infoMatrix[r1][c1].bottomZeroesCount >= height

    // If the top left point has more than or equal to amount
    // of consecutive zeroes in the right direction
    val isTopRowZeroes = infoMatrix[r1][c1].rightZeroesCount >= width

    // If the top right point has more than or equal to amount
    // of consecutive zeroes in the bottom direction
    val isRightColZeroes = infoMatrix[r1][c2].bottomZeroesCount >= height

    // If the bottom left point has more than or equal to amount
    // of consecutive zeroes in the right direction
    val isBottomRowZeroes = infoMatrix[r2][c1].rightZeroesCount >= width

    // If all of these are true then we have a square of zeroes.
    return isLeftColZeroes && isTopRowZeroes && isRightColZeroes &&
isBottomRowZeroes
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/countInversions/solution1/countInversions.kt

```
package ae.veryHard.countInversions.solution1

fun countInversions(array: MutableList<Int>): Int {
    return countSubarrayInversions(array, 0, array.size)
}

// Count inversion in a subarray bounded with left and right indexes
// Left is inclusive, right is exclusive
fun countSubarrayInversions(array: MutableList<Int>, left: Int, right: Int): Int {

    // If the array size is 1 or less then return 0.
    // An array of length 1 or 0 will have 0 inversions by definition.
    if (right - left <= 1)
        return 0

    // Get the mid pointer
    val mid = left + (right - left) / 2

    val leftInversions = countSubarrayInversions(array, left, mid)

    val rightInversions = countSubarrayInversions(array, mid, right)

    // Get number of inversions while merging the two halves.
    val mergedInversions = mergeAndCountInversions(array, left, mid, right)

    // Finally here we return inversions count in left, right and merge
    // step.
    return leftInversions + rightInversions + mergedInversions
}

// Count inversions while merging two *SORTED* subarrays
// right is exclusive
fun mergeAndCountInversions(array: MutableList<Int>, left: Int, mid: Int,
right: Int): Int {
    // idx1 starts at left and idx2 starts at mid.
    var idx1 = left
    var idx2 = mid

    // We assume inversions count is 0 and we declare the array which will
    // be used to merge the sorted arrays.
    var inversions = 0
    val sortedArray = mutableListOf<Int>()

    // While both subarrays has element
    while (idx1 < mid && idx2 < right) {
        // If the next number to insert is from the first subarray
        // e.g. number at first subarray's current pointer is less than or
        // equal to the number at second array's current pointer
        // Then there is no inversion here. Just insert the number in the
        // merged array.
        if (array[idx1] <= array[idx2]) {
            sortedArray.add(array[idx1])
            idx1++
        } else {
            sortedArray.add(array[idx2])
            idx2++
            inversions++
        }
    }

    // Add remaining elements of first subarray
    while (idx1 < mid) {
        sortedArray.add(array[idx1])
        idx1++
    }

    // Add remaining elements of second subarray
    while (idx2 < right) {
        sortedArray.add(array[idx2])
        idx2++
    }

    // Copy sorted array back to original array
    for (i in 0..sortedArray.size - 1) {
        array[i] = sortedArray[i]
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/countInversions/solution1/countInversions.kt

```
        sortedArray.add(array[idx1])
        idx1 += 1

        // Else the next number to insert is from the second subarray
        // e.g. number at second subarray's current pointer is less
        than the number at first array's current pointer
        // Then there are inversions
        // And the count of inversions here is simply the number of
        elements from the first subarray which are yet to be inserted in the merged
        array.
        // That is (end idx of first subarray + 1) - current idx in the
        first subarray
        // After incrementing inversion count with this value insert
        the number in the merged array.
    } else {
        inversions += (mid - idx1)
        sortedArray.add(array[idx2])
        idx2 += 1
    }
}

// Insert all remaining elements from the first subarray into the
merged array.
while (idx1 < mid) {
    sortedArray.add(array[idx1])
    idx1 += 1
}

// Insert all remaining elements from the second subarray into the
merged array.
while (idx2 < right) {
    sortedArray.add(array[idx2])
    idx2 += 1
}

// Copy the sorted elements into the corresponding segment of the main
array.
// Main array's current segment of operation starts at left
// So copy the sorted elements into the main array starting from `left
` index.
for (i in 0 .. sortedArray.lastIndex) {
    array[left + i] = sortedArray[i]
}

// Return accumulated count of inversions.
return inversions
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/detectArbitrage/solution1/detectArbitrage.kt

```
package ae.veryHard.detectArbitrage.solution1

import kotlin.math.log

fun detectArbitrage(exchangeRates: List<List<Double>>): Boolean {
    // Belman-Ford algorithm can detect negative weight cycle.
    // For a graph like below,
    //   1
    //   / \
    //   a   b
    //   \   /
    //   \ /
    //   2
    // An arbitrage will mean,
    //  $a * b > 1 = \log(a * b) > \log(1) = \log(a) + \log(b) > 0$ 
    // Which is a positive weight cycle. Belman-Ford algorithm is used to
    // detect negative weight cycles.
    // So we need to change it to  $-(\log(a) + \log(b)) < 0$ 
    // Or  $-\log(a) - \log(b) < 0$ 
    // Hence we need to convert all of the edge weights to their negative
    // logarithms.
    val logExchangeRates = convertToLogMatrix(exchangeRates)
    return foundNegativeWeightCycle(logExchangeRates, 0)
}

fun convertToLogMatrix(exchangeRates: List<List<Double>>): List<List<Double
>> {
    return exchangeRates.map { singleCurrencyRates ->
        singleCurrencyRates.map { -log(it, 10.0) }
    }
}

// Belman-Ford algorithm
fun foundNegativeWeightCycle(graph: List<List<Double>>, start: Int):
Boolean {
    val n = graph.size

    // Initially all of the nodes will have infinity distance
    // except the start node, which will have 0 distance
    val distances = MutableList(n) { Double.MAX_VALUE }
    distances[start] = 0.0

    // For a graph with n nodes from any source to any destination
    // the minimum path will contain at most  $(n - 1)$  edges.
    // Hence we need to perform relaxation  $(n - 1)$  times.
    // After  $(n - 1)$  steps all the nodes will reach their minimum distances
    // from the start node
    // And it will be stabilized.
    // If we don't have negative weighted cycle
    // then any further relaxation step won't change the distances anymore.
    for (i in 0 until n - 1) {
        // But if within  $(n - 1)$  steps relaxation
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/detectArbitrage/solution1/detectArbitrage.kt

```
// doesn't change distances anymore then
// prematurely we can say that the graph
// doesn't contain any negative weighted cycles.
// And any more relaxation will not change the distances anymore.
if (!relaxEdgesAndUpdateDistances(graph, distances))
    return false
}

// At this point the graph is stabilized.
// If we perform one more relaxation and the distances get updated
// Then we have a negative weighted cycle. So we will return true.
return relaxEdgesAndUpdateDistances(graph, distances)
}

fun relaxEdgesAndUpdateDistances(graph: List<List<Double>>, distances:
MutableList<Double>): Boolean {
    // We need to keep track if any distance got updated in this step.
    // Initially it is started as false.
    var updated = false

    // We consider every row as a source
    // In the row the elements themselves are edge weights
    graph.forEachIndexed { sourceIdx, edges ->
        // Now each edge index is destination index
        // And each element is a edge weight
        edges.forEachIndexed { destinationIdx, edgeWeight ->

            // If we use this source to arrive this destination using,
            // destination distance = current source's distance +
destination's distance from current source
            // If this distance is smaller than destination's current
distance
                // Then we need to update destination's current distance.
                if (distances[sourceIdx] + edgeWeight < distances[
destinationIdx]) {

                    distances[destinationIdx] = distances[sourceIdx] +
edgeWeight
                    // And we will mark updated to true, because we did update
a distance in this step.
                    updated = true
                }
            }
        }
    }

    // We will return if we update any distance in this step.
    return updated
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/apartmentHunting/solution1/apartmentHunting.kt

```
package ae.veryHard.apartmentHunting.solution1

import kotlin.math.*

fun apartmentHunting(blocks: List<Map<String, Boolean>>, reqs: List<String>): Int {
    val maxDistancesAtBlocks = mutableListOf(blocks.size) { Integer.MIN_VALUE
}

    for (i in 0 .. blocks.lastIndex) {
        reqs.forEach { req ->
            var minDistanceToRequirement = blocks
                .foldIndexed(Integer.MAX_VALUE) { j, minDistance,
currentBlock ->
                if (currentBlock[req]!!) {
                    min(minDistance, getDistance(i, j))
                } else minDistance
            }

            maxDistancesAtBlocks[i] = max(minDistanceToRequirement,
maxDistancesAtBlocks[i])
        }
    }

    return getIdxWithMinRequirement(maxDistancesAtBlocks)
}

fun getDistance(i: Int, j: Int) = abs(i - j)

fun getIdxWithMinRequirement(maxDistancesAtBlocks: List<Int>) : Int {
    val minElement = maxDistancesAtBlocks.minOrNull()!!
    return maxDistancesAtBlocks.indexOf(minElement)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/apartmentHunting/solution2/apartmentHunting.kt

```
package ae.veryHard.apartmentHunting.solution2

import kotlin.math.abs
import kotlin.math.min

fun apartmentHunting(blocks: List<Map<String, Boolean>>, reqs: List<String>): Int {
    val minDistancesToRequirements = getMinimumDistancesToRequirements(
        blocks, reqs)
    val maxDistanceAtBlocks = getMaxDistanceAtBlocks(
        minDistancesToRequirements)
    return getIdxWithMinRequirement(maxDistanceAtBlocks)
}

// For each requirement create an array of minimum distances from a block
// to that requirement
// So returned value from this function will be array of such distances
// array for all requirements
fun getMinimumDistancesToRequirements(blocks: List<Map<String, Boolean>>,
reqs: List<String>) : List<List<Int>> {
    val minimumDistancesToRequirements = reqs.map { req ->
        getMinimumDistancesToRequirement(blocks, req)
    }

    return minimumDistancesToRequirements
}

// Visit the array from left to right and from right to left to
// calculate minimum distance to requirement
fun getMinimumDistancesToRequirement(blocks: List<Map<String, Boolean>>,
req: String) : List<Int> {
    val minDistancesToRequirement = mutableListOf(blocks.size) { Int.MAX_VALUE }

    // Assign it to INT_MAX because INT_MAX is the highest possible number.
    // See below comments for clarity
    var lastIdxOfRequirement = Integer.MAX_VALUE

    // Left to right partial calculation
    for (idx in 0 .. blocks.lastIndex) {
        val currentBlock = blocks[idx]

        // If index 0 doesn't contain requirement then
        minDistancesToRequirement[0] will be INT_MAX - 0
        // minDistancesToRequirement[1] will be INT_MAX - 1 and so on.
        // Which are also big numbers
        // But for example if we used 0 to initialize lastRequirementIdx
        then these values would be 0, -1, -2 etc
        // Which are smaller than other legit distances.
        // As a result line 78 will fail. Hence we need to initialize
        lastRequirementIdx to INT_MAX.
        if (currentBlock[req]!!) {
            lastIdxOfRequirement = idx
        }
    }
}
```

```
File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/apartmentHunting/solution2/apartmentHunting.kt
}

    minDistancesToRequirement[idx] = getDistance(idx,
lastIdxOfRequirement)
}

// Right to left partial calculation
for (idx in blocks.lastIndex downTo 0) {
    val currentBlock = blocks[idx]

    if (currentBlock[req]!!) {
        lastIdxOfRequirement = idx
    }

    minDistancesToRequirement[idx] = min(minDistancesToRequirement[idx],
getDistance(idx, lastIdxOfRequirement))
}

return minDistancesToRequirement
}

// Group elements of each requirements to their maximum value for every
block.
fun getMaxDistanceAtBlocks(minDistancesToRequirements: List<List<Int>>): List<Int> {
    val blockCount = minDistancesToRequirements[0].size

    val maxDistances = mutableListOf<Int>()

    for (i in 0 until blockCount) {

        val distancesAtBlock = minDistancesToRequirements.map { it[i] }
        maxDistances.add(distancesAtBlock.maxOrNull()!!)
    }

    return maxDistances
}

fun getDistance(i: Int, j: Int) = abs(i - j)

// Get index of the minimum value.
fun getIdxWithMinRequirement(maxDistancesAtBlocks: List<Int>) : Int {
    val minElement = maxDistancesAtBlocks.minOrNull()!!
    return maxDistancesAtBlocks.indexOf(minElement)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/calendarMatching/solution1/calendarMatching.kt

```
package ae.veryHard.calendarMatching.solution1

import kotlin.math.max

fun calendarMatching(calendar1: List<List<String>>, dailyBounds1: List<String>, calendar2: List<List<String>>, dailyBounds2: List<String>, meetingDuration: Int): List<List<String>> {
    val updatedCalendar1 = updateCalendar(calendar1, dailyBounds1)
    val updatedCalendar2 = updateCalendar(calendar2, dailyBounds2)

    val mergedCalendar = mergeCalendars(updatedCalendar1, updatedCalendar2)
    val mergedIntervals = mergeIntervals(mergedCalendar)
    val availableSlots = getAvailableSlots(mergedIntervals, meetingDuration)
}

return availableSlots.map { it.toListBounds() }
}

fun updateCalendar(calendar: List<List<String>>, dailyBound: List<String>): List<IntMeeting> {
    val updatedCalendar = mutableListOf(StringMeeting("00:00", dailyBound[0]))
    calendar.forEach { meeting ->
        updatedCalendar.add(StringMeeting(meeting[0], meeting[1]))
    }
    updatedCalendar.add(StringMeeting(dailyBound[1], "23:59"))

    return updatedCalendar.map { it.toIntMeeting() }
}

fun mergeCalendars(calendar1: List<IntMeeting>, calendar2: List<IntMeeting>): List<IntMeeting> {
    val mergedCalendar = mutableListOf<IntMeeting>()

    var p1 = 0
    var p2 = 0

    while (p1 < calendar1.size && p2 < calendar2.size) {
        if (calendar1[p1].start < calendar2[p2].start) {
            mergedCalendar.add(calendar1[p1])
            p1 += 1
        } else {
            mergedCalendar.add(calendar2[p2])
            p2 += 1
        }
    }

    while (p1 < calendar1.size) {
        mergedCalendar.add(calendar1[p1])
        p1++
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/calendarMatching/solution1/calendarMatching.kt

```
while (p2 < calendar2.size) {
    mergedCalendar.add(calendar2[p2])
    p2++
}

return mergedCalendar
}

fun mergeIntervals(calendar: List<IntMeeting>): List<IntMeeting> {
    val mergedIntervals = mutableListOf<IntMeeting>()
    mergedIntervals.add(calendar.first())

    for (i in 1 .. calendar.lastIndex) {
        val currentMeeting = calendar[i]

        if (currentMeeting.start > mergedIntervals.last().end) {
            mergedIntervals.add(currentMeeting)
        } else {
            mergedIntervals.last().end = max(currentMeeting.end,
mergedIntervals.last().end)
        }
    }

    return mergedIntervals
}

fun getAvailableSlots(calendar: List<IntMeeting>, meetingDuration: Int): List<IntMeeting> {

    val availableSlots = mutableListOf<IntMeeting>()

    for (i in 1 .. calendar.lastIndex) {
        val currentMeeting = calendar[i]
        val lastMeeting = calendar[i - 1]

        if (currentMeeting.start - lastMeeting.end >= meetingDuration) {
            availableSlots.add(IntMeeting(lastMeeting.end, currentMeeting.
start))
        }
    }

    return availableSlots
}

data class StringMeeting(val start: String, val end: String) {
    fun toIntMeeting() = IntMeeting(toIntTime(start), toIntTime(end))
}

data class IntMeeting(var start: Int, var end: Int) {
    private fun toStringMeeting() = StringMeeting(toStringTime(start),
toStringTime(end))
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/calendarMatching/solution1/calendarMatching.kt

```
fun toListBounds(): List<String> {
    val stringMeeting = toStringMeeting()
    return listOf(stringMeeting.start, stringMeeting.end)
}

fun toIntTime(time: String): Int {
    val parts = time.split(":")
    val hour = parts[0].toInt()
    val minute = parts[1].toInt()
    return hour * 60 + minute
}

fun toStringTime(time: Int): String {
    val hour = time / 60
    val minute = time % 60

    val hourStr = hour.toString()
    val minuteStr = if (minute < 10) "0$minute" else minute.toString()

    return "$hourStr:$minuteStr"
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/rightSiblingTree/solution1/rightSiblingTree.kt

```
package ae.veryHard.rightSiblingTree.solution1

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun rightSiblingTree(tree: BinaryTree): BinaryTree {
    mutate(tree, null, false)
    return tree
}

// There are two patterns:
// if a node is the left child of another node,
//   its right sibling is its parent's right child;
// if a node is the right child of another node,
//   its right sibling is its parent's right sibling's left child.
fun mutate(node: BinaryTree?, parent: BinaryTree?, isLeftChild: Boolean) {
    node ?: return

    val left = node.left
    val right = node.right

    // We transform left child first.
    mutate(left, node, true)

    // Then we transform the node.
    if (parent == null) {
        node.right = null
    } else if (isLeftChild) {
        // At this point parent->right
        // contains original right child of parent.
        // Because we transform left child before
        // transforming the parent itself.
        node.right = parent.right
    } else {
        // parent->right can be null when
        // parent->right was modified before to be null.
        // Because we transform parent before
        // transforming its right child.
        // So parent's right will be overwritten
        // before we will reach to its right child.
        if (parent.right == null) {
            node.right = null
        } else {
            // At this point parent->right
            // is the modified right pointer to its right sibling.
            // So parent->right->left is the
            // left child of the parent's right sibling.
            node.right = parent.right!!.left
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/rightSiblingTree/solution1/rightSiblingTree.kt

```
}

// Then we transform the right child.
mutate(right, node, false)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/rightSmallerThan/solution1/rightSmallerThan.kt

```
package ae.veryHard.rightSmallerThan.solution1

// Construct a BST by inserting the input array's integers
// one by one, in reverse order (from right to left).
// At each insertion, once a new BST node is positioned in the BST,
// the number of nodes in its parent node's left subtree
// (plus the parent node itself, if its value is smaller than the inserted
// node's value)
// is the number of "right-smaller-than" elements for the element being
// inserted.
fun rightSmallerThan(array: List<Int>): List<Int> {
    if (array.isEmpty()) return listOf()

    val root = SpecialBST(array.lastIndex, array.last())

    for (i in array.lastIndex - 1 downTo 0) {
        root.insert(i, array[i])
    }

    val result = mutableListOf(array.size) { 0 }
    populateRightSmallerThan(root, result)

    return result
}

class SpecialBST(val idx: Int, val value: Int, val
smallerCountInCreationTime: Int = 0) {
    var left: SpecialBST? = null
    var right: SpecialBST? = null

    var leftChildrenCount = 0

    fun insert(idx: Int, value: Int, smallerCountInCreationTime: Int = 0) {

        if (value < this.value) {
            leftChildrenCount += 1

            if (left == null) {
                left = SpecialBST(idx, value, smallerCountInCreationTime)
            } else {
                left!!.insert(idx, value, smallerCountInCreationTime)
            }
        } else {
            var nextSmallerCountInCreationTime = smallerCountInCreationTime
            nextSmallerCountInCreationTime += leftChildrenCount
            if (value > this.value) {
                nextSmallerCountInCreationTime += 1
            }

            if (right == null) {
                right = SpecialBST(idx, value,
                    nextSmallerCountInCreationTime)
            } else {
                right.insert(idx, value, nextSmallerCountInCreationTime)
            }
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/rightSmallerThan/solution1/rightSmallerThan.kt

```
nextSmallerCountInCreationTime)
    } else {
        right!!.insert(idx, value, nextSmallerCountInCreationTime)
    }
}
}

fun populateRightSmallerThan(node: SpecialBST?, result: MutableList<Int>) {
    node ?: return

    result[node.idx] = node.smallerCountInCreationTime

    populateRightSmallerThan(node.left, result)
    populateRightSmallerThan(node.right, result)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/rightSmallerThan/solution2/rightSmallerThan.kt

```
package ae.veryHard.rightSmallerThan.solution2

fun rightSmallerThan(array: List<Int>): List<Int> {
    if (array.isEmpty()) return listOf()

    val root = SpecialBST(array.lastIndex, array.last())

    val result = mutableListOf(array.size) { 0 }
    for (i in array.lastIndex - 1 downTo 0) {
        root.insert(i, array[i], result)
    }

    return result
}

class SpecialBST(val idx: Int, val value: Int) {
    var left: SpecialBST? = null
    var right: SpecialBST? = null

    var leftChildrenCount = 0

    fun insert(idx: Int, value: Int, result: MutableList<Int>,
smallerCountInCreationTime: Int = 0) {

        if (value < this.value) {
            leftChildrenCount += 1

            if (left == null) {
                left = SpecialBST(idx, value)
                result[idx] = smallerCountInCreationTime
            } else {
                left!!.insert(idx, value, result,
smallerCountInCreationTime)
            }
        } else {
            var nextSmallerCountInCreationTime = smallerCountInCreationTime
            nextSmallerCountInCreationTime += leftChildrenCount
            if (value > this.value) {
                nextSmallerCountInCreationTime += 1
            }

            if (right == null) {
                right = SpecialBST(idx, value)
                result[idx] = nextSmallerCountInCreationTime
            } else {
                right!!.insert(idx, value, result,
nextSmallerCountInCreationTime)
            }
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/waterfallStreams/solution1/waterfallStreams.kt

```
package ae.veryHard.waterfallStreams.solution1

fun waterfallStreams(array: List<List<Double>>, source: Int): List<Double>
> {
    // First row will contain full water.
    // It is the row where the water will pour into.
    // So we don't have to do any work here other than marking the source.
    // rowAbove is a copy array because we don't want to modify the
    original array.
    var rowAbove = ArrayList(array[0])
    rowAbove[source] = -1.0

    // We have to operate starting from second row
    // because the first row will have the source.
    for (i in 1 .. array.lastIndex) {

        // Current row is also a copy array, because here too
        // we don't want to modify the original array.
        val currentRow = ArrayList(array[i])

        // For every item in current row
        for (j in 0 .. currentRow.lastIndex) {

            // If the above cell doesn't contain any water then continue.
            if (rowAbove[j] >= 0) {
                continue
            }

            // If current row is not a block space
            // then water will pour into it from the above cell.
            if (currentRow[j] != 1.0) {
                currentRow[j] += rowAbove[j]
                // After pouring the water we can continue.
                continue
            }

            // Otherwise current cell is an obstacle.
            // Water stream will split into two halves here.
            val splitWater = rowAbove[j] / 2.0

            // Go to the left empty cell where water can pour into
            var left = j
            // Inside the loop we will use index left - 1.
            while (left - 1 >= 0) {
                left -= 1

                // Water is blocked by the cell above. This part of water
                is wasted.
                if (rowAbove[left] == 1.0) {
                    break
                }
            }
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/waterfallStreams/solution1/waterfallStreams.kt

```
// If current cell at left index is empty space (not a
block)
    // then water will pour into it.
    if (currentRow[left] <= 0) {
        currentRow[left] += splitWater
        // Then we don't need to go any more left,
        // because water got poured already.
        break
    }
}

// Go to the right empty cell where water can pour into
var right = j
// Inside the loop we will use index right + 1.
while (right + 1 < currentRow.size) {
    right += 1

    // Water is blocked by the cell above. This part of water
is wasted.
    if (rowAbove[right] == 1.0) {
        break
    }

    // If current cell at right index is empty space (not a
block)
    // then water will pour into it.
    if (currentRow[right] <= 0) {
        currentRow[right] += splitWater
        // Then we don't need to go any more right,
        // because water got poured already.
        break
    }
}

rowAbove = currentRow
}

// At this point rowAbove contains the last row
// It's elements are stored as negative fraction
// It doesn't contain any block (1 value)
return rowAbove.map { it * -100 }
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/flattenBinaryTree/solution1/flattenBinaryTree.kt

```
package ae.veryHard.flattenBinaryTree.solution1

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun flattenBinaryTree(root: BinaryTree): BinaryTree {
    val inorderList = mutableListOf<BinaryTree>()

    populateInorderList(root, inorderList)

    for (i in 0 until inorderList.lastIndex) {
        val prevNode = inorderList[i]
        val nextNode = inorderList[i + 1]

        prevNode.right = nextNode
        nextNode.left = prevNode
    }

    return inorderList[0]
}

fun populateInorderList(node: BinaryTree?, inorderList: MutableList<BinaryTree>) {
    node ?: return

    populateInorderList(node.left, inorderList)
    inorderList.add(node)
    populateInorderList(node.right, inorderList)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/flattenBinaryTree/solution2/flattenBinaryTree.kt

```
package ae.veryHard.flattenBinaryTree.solution2

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun flattenBinaryTree(root: BinaryTree): BinaryTree {
    return flattenTree(root).first
}

fun flattenTree(node: BinaryTree): Pair<BinaryTree, BinaryTree> {
    val leftNode = if (node.left == null) {
        node
    } else {
        val (leftMostInLeftSubtree, rightMostInLeftSubtree) = flattenTree(
            node.left!!)
        connectNodes(rightMostInLeftSubtree, node)
        leftMostInLeftSubtree
    }

    val rightNode = if (node.right == null) {
        node
    } else {
        val (leftMostInRightSubtree, rightMostInRightSubtree) = flattenTree(
            node.right!!)
        connectNodes(node, leftMostInRightSubtree)
        rightMostInRightSubtree
    }

    return leftNode to rightNode
}

fun connectNodes(node1: BinaryTree, node2: BinaryTree) {
    node1.right = node2
    node2.left = node1
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/lineThroughPoints/solution1/lineThroughPoints.kt

```
package ae.veryHard.lineThroughPoints.solution1

import kotlin.math.max

fun lineThroughPoints(points: List<List<Int>>): Int {
    var maxPointsThroughLines = 1

    points.forEachIndexed { i, p0 ->

        // Create a brand new map to store slopes only for
        // lines which include this P0.
        // In this way if we find multiple pairs
        // having the same slope we know that all those points
        // lie on the same line because they all
        // pass through a common point p0.
        val slopes = mutableMapOf<String, Int>()

        for (j in i + 1 .. points.lastIndex) {

            val p1 = points[j]

            val slope = getSlopeBetweenPoints(p0, p1)
            val (rise, run) = slope
            val slopeKey = getSlopeKey(rise, run)

            if (!slopes.containsKey(slopeKey)) {
                slopes[slopeKey] = 1
            }
            slopes[slopeKey] = slopes[slopeKey]!! + 1
        }

        getMaxValueFromMap(slopes)?.let { maxValue ->
            maxPointsThroughLines = max(maxPointsThroughLines, maxValue)
        }
    }

    return maxPointsThroughLines
}

fun getMaxValueFromMap(slopes: Map<String, Int>) =
    slopes.values.maxOrNull()

fun getSlopeBetweenPoints(p0: List<Int>, p1: List<Int>): List<Int> {
    val (p0X, p0Y) = p0
    val (p1X, p1Y) = p1

    var rise = p1Y - p0Y
    var run = p1X - p0X
    val gcd = findGcd(rise, run)

    rise /= gcd
    run /= gcd
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/lineThroughPoints/solution1/lineThroughPoints.kt

```
if (run < 0) {
    return listOf(-rise, -run)
}

return listOf(rise, run)
}

fun findGcd(a: Int, b: Int): Int {
    if (a == 0) return b
    if (b == 0) return a
    return findGcd(b, a % b)
}

fun getSlopeKey(rise: Int, run: Int) = "$rise|$run"
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/mergeSortedArrays/solution1/mergeSortedArrays.kt

```
package ae.veryHard.mergeSortedArrays.solution1

import java.util.PriorityQueue

fun mergeSortedArrays(arrays: List<List<Int>>): List<Int> {
    val sortedList = mutableListOf<Int>()

    val heap = PriorityQueue<Item> { item1, item2 ->
        item1.elementValue - item2.elementValue
    }

    arrays.forEachIndexed { idx, array ->
        heap.add(Item(idx, 0, array[0]))
    }

    while (heap.isNotEmpty()) {

        val currentItem = heap.poll()

        val (arrayIdx, elementIdx, elementValue) = currentItem

        sortedList.add(elementValue)

        if (elementIdx < arrays[arrayIdx].lastIndex) {
            heap.add(Item(arrayIdx, elementIdx + 1, arrays[arrayIdx][elementIdx + 1]))
        }
    }

    return sortedList
}

data class Item(val arrayIdx: Int, val elementIdx: Int, val elementValue: Int)
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/airportConnections/solution1/airportConnections.kt

```
package ae.veryHard.airportConnections.solution1

fun airportConnections(airports: List<String>, routes: List<Pair<String, String>>, startingAirport: String): Int {
    val graph = buildGraph(airports, routes)
    val unreachableAirports = populateUnreachableAirports(graph, airports, startingAirport)
    markUnreachableAirportsWithAirportsTheyUnlock(graph, unreachableAirports)
    return getNumberOfNewConnectionsRequired(graph, unreachableAirports)
}

data class AirportNode(val airport: String) {
    val connections = mutableListOf<String>()
    var isReachable = true
    var unlocksUnreachableAirports = mutableListOf<String>()
}

fun buildGraph(airports: List<String>, routes: List<Pair<String, String>>): MutableMap<String, AirportNode> {
    val graph = airports.map { airport ->
        airport to AirportNode(airport)
    }.toMap().toMutableMap()

    for ((airport, connection) in routes) {
        graph[airport]!!.connections.add(connection)
    }

    return graph
}

fun populateUnreachableAirports(graph: MutableMap<String, AirportNode>, airports: List<String>, startingAirport: String): List<AirportNode> {
    val startingNode = graph[startingAirport]!!
    val visited = mutableSetOf<String>()

    dfsVisitAirports(startingNode, graph, visited)

    return airports.filter { airport ->
        !visited.contains(airport)
    }.map {
        graph[it]!!.apply {
            isReachable = false
        }
    }
}

fun markUnreachableAirportsWithAirportsTheyUnlock(graph: MutableMap<String, AirportNode>, unreachableAirports: List<AirportNode>) {
    for (unreachableAirport in unreachableAirports) {
        val visited = mutableSetOf<String>()
        dfsPopulateUnlockedUnreachableNodes(unreachableAirport, graph,
    
```

```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/airportConnections/solution1/airportConnections.kt

visited)
    unreachableAirport.unlocksUnreachableAirports = visited.
toMutableList()
}
}

fun dfsVisitAirports(node: AirportNode, graph: MutableMap<String, AirportNode>, visited: MutableSet<String>) {
    val airport = node.airport
    if (visited.contains(airport)) return
    visited.add(airport)
    for (connection in node.connections) {
        dfsVisitAirports(graph[connection]!!, graph, visited)
    }
}

fun dfsPopulateUnlockedUnreachableNodes(node: AirportNode, graph: MutableMap<String, AirportNode>, visited: MutableSet<String>) {
    val airport = node.airport
    if (visited.contains(airport)) return
    if (node.isReachable) return
    visited.add(airport)
    for (connection in node.connections) {
        dfsPopulateUnlockedUnreachableNodes(graph[connection]!!, graph,
visited)
    }
}

fun getNumberOfNewConnectionsRequired(graph: MutableMap<String, AirportNode>, unreachableAirportNodes: List<AirportNode>): Int {
    val sortedUnreachableAirports = unreachableAirportNodes.sortedWith(
Comparator<AirportNode> { airportNode1, airportNode2 -
    airportNode2.unlocksUnreachableAirports.size.compareTo(airportNode1.
.unlocksUnreachableAirports.size)
})

    var numberofNewConnectionsRequired = 0

    for (unreachableAirportNode in sortedUnreachableAirports) {
        if (unreachableAirportNode.isReachable) continue
        numberofNewConnectionsRequired += 1

        for (unlockedUnreachableAirport in unreachableAirportNode.
unlocksUnreachableAirports) {
            graph[unlockedUnreachableAirport]!!.isReachable = true
        }
    }

    return numberofNewConnectionsRequired
}

```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/nonAttackingQueens/solution1/nonAttackingQueens.kt

```
package ae.veryHard.nonAttackingQueens.solution1

import kotlin.math.abs

// Time: O(n!) Lower Bound
fun nonAttackingQueens(n: Int): Int {
    val queenPlacements = MutableList(n) { 0 }
    return getNonAttackingQueenPlacements(0, queenPlacements, n)
}

fun getNonAttackingQueenPlacements(row: Int, queenPlacements: MutableList<Int>, boardSize: Int): Int {
    // If we reached past number of total rows, then we are good, we have a
    // non attacking queen placement.
    if (row == boardSize)
        return 1

    var nonAttackingPlacementCount = 0

    // For all column attempt to place the queen
    for (col in 0 until boardSize) {

        // If the queen placement in this column is non-attacking
        if (isNonAttackingPlacement(row, col, queenPlacements)) {

            // Place the queen in this col for this row
            queenPlacements[row] = col

            // Then try to place queen in next row and accumulate the "
            // bubbled-up" non attacking placements for next row
            nonAttackingPlacementCount += getNonAttackingQueenPlacements(
                row + 1, queenPlacements, boardSize)
        }
    }

    return nonAttackingPlacementCount
}

// Check if the queen placement in row and col is a not attacking placement
fun isNonAttackingPlacement(row: Int, col: Int, queenPlacements:
MutableList<Int>): Boolean {

    // To determine if a placement is non-attacking,
    // we need to examine all the row above for potential conflicts
    for (r in 0 until row) {

        val placementColInRow = queenPlacements[r]

        val blockedByColumn = placementColInRow == col

        // Diagonal check by abs(r1 - r2) == abs(c1 - c2)
        val blockedByDiagonal = abs(placementColInRow - col) == row - r
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/nonAttackingQueens/solution1/nonAttackingQueens.kt

```
// If we get any conflict either in column or in diagonal, we then
return false
    if (blockedByColumn || blockedByDiagonal)
        return false
}

return true
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/nonAttackingQueens/solution2/nonAttackingQueens.kt

```
package ae.veryHard.nonAttackingQueens.solution2

// Time: O(n!) Upper Bound
fun nonAttackingQueens(n: Int): Int {
    val blockedColumns = mutableSetOf<Int>()
    val blockedUpDiagonals = mutableSetOf<Int>()
    val blockedDownDiagonals = mutableSetOf<Int>()
    return getNonAttackingQueenPlacements(0, blockedColumns,
    blockedUpDiagonals, blockedDownDiagonals, n)
}

fun getNonAttackingQueenPlacements(row: Int, blockedColumns: MutableSet<Int>,
>, blockedUpDiagonals: MutableSet<Int>, blockedDownDiagonals: MutableSet<
Int>, boardSize: Int): Int {
    // If we reached past number of total rows, then we are good, we have a
    non attacking queen placement.
    if (row == boardSize)
        return 1

    var nonAttackingPlacementCount = 0

    // For all column attempt to place the queen
    for (col in 0 until boardSize) {

        // If the queen placement in this column is non-attacking
        if (isNonAttackingPlacement(row, col, blockedColumns,
        blockedUpDiagonals, blockedDownDiagonals)) {

            // Place the queen in this col for this row
            placeQueen(row, col, blockedColumns, blockedUpDiagonals,
            blockedDownDiagonals)

            // Then try to place queen in next row and accumulate the "
            bubbled-up" non attacking placements for next row
            nonAttackingPlacementCount += getNonAttackingQueenPlacements(
            row + 1, blockedColumns, blockedUpDiagonals, blockedDownDiagonals,
            boardSize)

            // Backtrack, we need to clear the sets for current placement.
            // Otherwise for future placements we may mark future columns
            as blocked when they are not relevant (not blocked) anymore.
            removeQueen(row, col, blockedColumns, blockedUpDiagonals,
            blockedDownDiagonals)
        }
    }

    return nonAttackingPlacementCount
}

// Check if the queen placement in row and col is a not attacking placement
fun isNonAttackingPlacement(row: Int, col: Int, blockedColumns: MutableSet<
Int>, blockedUpDiagonals: MutableSet<Int>, blockedDownDiagonals: MutableSet
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/nonAttackingQueens/solution2/nonAttackingQueens.kt

```
<Int>): Boolean {
    if (blockedColumns.contains(col)) {
        return false
    }

    if (blockedUpDiagonals.contains(row + col)) {
        return false
    }

    if (blockedDownDiagonals.contains(row - col)) {
        return false
    }

    return true
}

fun placeQueen(row: Int, col: Int, blockedColumns: MutableSet<Int>,
    blockedUpDiagonals: MutableSet<Int>, blockedDownDiagonals: MutableSet<Int
    >) {
    blockedColumns.add(col)
    blockedUpDiagonals.add(row + col)
    blockedDownDiagonals.add(row - col)
}

fun removeQueen(row: Int, col: Int, blockedColumns: MutableSet<Int>,
    blockedUpDiagonals: MutableSet<Int>, blockedDownDiagonals: MutableSet<Int
    >) {
    blockedColumns.remove(col)
    blockedUpDiagonals.remove(row + col)
    blockedDownDiagonals.remove(row - col)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/longestStringChains/solution1/longestStringChains.kt

```
package ae.veryHard.longestStringChains.solution1

fun longestStringChain(strings: List<String>): List<String> {
    // OR
    // val sortedStrings = strings.sortedBy { it.length }
    // OR
    /* val sortedStrings = strings.sortedWith(Comparator<String> { str1,
str2 ->
        str1.length - str2.length
    }) */
    // OR
    /* val sortedStrings = strings.sortedWith(Comparator<String> { str1,
str2 ->
        when {
            str1.length > str2.length -> 1
            str1.length < str2.length -> -1
            else -> 0
        }
    }) */
}

/*
New Syntax - Lambda outside, no parenthesis
val sortedStrings = strings.sortedWith { str1, str2 ->
    str1.length.compareTo(str2.length)
}
*/

// Sort the strings array by length for ease in implementation. Sorting
guarantees that
// the shorter string is computed before the present longer string
val sortedStrings = strings.sortedWith(Comparator<String> { str1, str2
->
    str1.length.compareTo(str2.length)
})

// Iterate through each string and prepare the map of string ->
stringChain object
val stringChains = sortedStrings.associateWith {
    StringChain("", 1)
}.toMutableMap()

// Find the longest string chain for each string from shortest to
longest string
sortedStrings.forEach { str ->
    findLongestStringChain(str, stringChains)
}

// Find the overall longest string chain, build the sequence and return
the chain
return buildLongestStringChains(strings, stringChains)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/longestStringChains/solution1/longestStringChains.kt

```
data class StringChain(var nextStringInChain: String, var stringChainLen: Int)

fun findLongestStringChain(str: String, stringChains: MutableMap<String, StringChain>) {
    for (i in 0 .. str.lastIndex) {
        val smallerSubstring = getSmallerString(str, i)
        if (!stringChains.containsKey(smallerSubstring)) continue
        updateLongestStringChain(str, smallerSubstring, stringChains)
    }
}

fun getSmallerString(str: String, idx: Int) = str.substring(0 until idx) + str.substring(idx + 1)

fun updateLongestStringChain(currentString: String, nextString: String, stringChains: MutableMap<String, StringChain>) {
    val currentStringChainLen = stringChains[currentString]!!.stringChainLen
    val nextStringChainLen = stringChains[nextString]!!.stringChainLen

    // Does using this next string makes the chain longer?
    if (nextStringChainLen + 1 > currentStringChainLen) {
        stringChains[currentString]?.nextStringInChain = nextString
        stringChains[currentString]?.stringChainLen = nextStringChainLen +
    1
    }
}

fun buildLongestStringChains(strings: List<String>, stringChains: Map<String, StringChain>): List<String> {
    var longestChainStartStr = ""
    var longestChainLen = 0

    for (str in strings) {
        val strChain = stringChains[str]!!
        if (strChain.stringChainLen > longestChainLen) {
            longestChainStartStr = str
            longestChainLen = strChain.stringChainLen
        }
    }

    val longestChain = mutableListOf<String>()
    var currentStr = longestChainStartStr

    while (currentStr != "") {
        longestChain.add(currentStr)
        currentStr = stringChains[currentStr]!!.nextStringInChain
    }

    return if (longestChain.size == 1) listOf() else longestChain
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/longestStringChains/solution1/longestStringChains.kt

}

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/rearrangeLinkedList/solution1/rearrangeLinkedList.kt

```
package ae.veryHard.rearrangeLinkedList.solution1

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun rearrangeLinkedList(head: LinkedList, k: Int): LinkedList {
    var smallerListHead: LinkedList? = null
    var smallerListTail: LinkedList? = null
    var equalListHead: LinkedList? = null
    var equalListTail: LinkedList? = null
    var greaterListHead: LinkedList? = null
    var greaterListTail: LinkedList? = null

    var current: LinkedList? = head

    while (current != null) {

        if (current.value < k) {
            val smallerListNodes = growLinkedList(smallerListHead,
smallerListTail, current)
            smallerListHead = smallerListNodes.first
            smallerListTail = smallerListNodes.second
        } else if (current.value == k) {
            val equalListNodes = growLinkedList(equalListHead,
equalListTail, current)
            equalListHead = equalListNodes.first
            equalListTail = equalListNodes.second
        } else {
            val greaterListNodes = growLinkedList(greaterListHead,
greaterListTail, current)
            greaterListHead = greaterListNodes.first
            greaterListTail = greaterListNodes.second
        }

        val prev = current
        current = current.next
        prev.next = null
    }

    val (firstConnectedListHead, firstConnectedListTail) =
connectLinkedLists(smallerListHead, smallerListTail, equalListHead,
equalListTail)
    val (secondConnectedListHead, _) = connectLinkedLists(
firstConnectedListHead, firstConnectedListTail, greaterListHead,
greaterListTail)

    return secondConnectedListHead!!
}

fun growLinkedList(head: LinkedList?, tail: LinkedList?, node: LinkedList
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/rearrangeLinkedList/solution1/rearrangeLinkedList.kt

```
?): Pair<LinkedList?, LinkedList?> {
    var newHead: LinkedList? = head
    val newTail: LinkedList? = node

    if (newHead == null) {
        newHead = node
    }

    if (tail != null) {
        tail.next = newTail
    }

    return newHead to newTail
}

fun connectLinkedLists(firstHead: LinkedList?, firstTail: LinkedList?,
secondHead: LinkedList?, secondTail: LinkedList?): Pair<LinkedList?, LinkedList?> {
    val mergedHead = firstHead ?: secondHead
    val mergedTail = secondTail ?: firstTail

    if (firstTail != null) {
        firstTail.next = secondHead
    }

    return mergedHead to mergedTail
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/allKindsOfNodeDepths/solution1/allKindsOfNodeDepths.kt

```
package ae.veryHard.allKindsOfNodeDepths.solution1

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

// O(n * log(n)) Time | O(h) Space
fun allKindsOfNodeDepths(root: BinaryTree): Int {
    val leftSumOfNodeDepths = root.left?.let { leftChild ->
        allKindsOfNodeDepths(leftChild)
    } ?: 0

    val rightSumOfNodeDepths = root.right?.let { rightChild ->
        allKindsOfNodeDepths(rightChild)
    } ?: 0

    return leftSumOfNodeDepths + rightSumOfNodeDepths + getNodeDepth(root)
}

fun getNodeDepth(node: BinaryTree?, depth: Int = 0): Int {
    node ?: return 0

    return depth + getNodeDepth(node.left, depth + 1) + getNodeDepth(node.
right, depth + 1)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/allKindsOfNodeDepths/solution2/allKindsOfNodeDepths.kt

```
package ae.veryHard.allKindsOfNodeDepths.solution2

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

// O(n) Time | O(n) Space
fun allKindsOfNodeDepths(root: BinaryTree): Int {
    val nodeCounts = mutableMapOf<BinaryTree, Int>()
    val nodeDepths = mutableMapOf<BinaryTree, Int>()

    populateNodeCounts(root, nodeCounts)
    populateNodeDepths(root, nodeDepths, nodeCounts)

    return getSumOfAllNodeDepths(root, nodeDepths)
}

fun populateNodeCounts(node: BinaryTree, nodeCounts: MutableMap<BinaryTree, Int>) {
    var count = 1

    if (node.left != null) {
        populateNodeCounts(node.left!!, nodeCounts)
        count += nodeCounts[node.left!!]!!
    }

    if (node.right != null) {
        populateNodeCounts(node.right!!, nodeCounts)
        count += nodeCounts[node.right!!]!!
    }

    nodeCounts[node] = count
}

fun populateNodeDepths(node: BinaryTree, nodeDepths: MutableMap<BinaryTree, Int>, nodeCounts: MutableMap<BinaryTree, Int>) {
    var depth = 0

    if (node.left != null) {
        populateNodeDepths(node.left!!, nodeDepths, nodeCounts)
        depth += (nodeDepths[node.left!!]!! + nodeCounts[node.left!!]!!)
    }

    if (node.right != null) {
        populateNodeDepths(node.right!!, nodeDepths, nodeCounts)
        depth += (nodeDepths[node.right!!]!! + nodeCounts[node.right!!]!!)
    }

    nodeDepths[node] = depth
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/allKindsOfNodeDepths/solution2/allKindsOfNodeDepths.kt

```
fun getSumOfAllNodeDepths(node: BinaryTreeNode?, nodeDepths: MutableMap<BinaryTreeNode, Int>): Int {
    node ?: return 0

    return nodeDepths[node]!! + getSumOfAllNodeDepths(node.left, nodeDepths)
} + getSumOfAllNodeDepths(node.right, nodeDepths)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/allKindsOfNodeDepths/solution3/allKindsOfNodeDepths.kt

```
package ae.veryHard.allKindsOfNodeDepths.solution3

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

// O(n) Time | O(h) Space
fun allKindsOfNodeDepths(root: BinaryTree): Int {
    return getNodeInfo(root).allNodeDepths
}

data class NodeInfo(val nodeCount: Int, val nodeDepths: Int, val
allNodeDepths: Int)

fun getNodeInfo(node: BinaryTree?): NodeInfo {
    node ?: return NodeInfo(0, 0, 0)

    val (leftNodeCount, leftNodeDepths, leftAllNodeDepths) = getNodeInfo(
node.left)
    val (rightNodeCount, rightNodeDepths, rightAllNodeDepths) = getNodeInfo(
node.right)

    val nodeCount = leftNodeCount + rightNodeCount + 1
    val nodeDepths = leftNodeDepths + leftNodeCount + rightNodeDepths +
rightNodeCount
    val allNodeDepths = nodeDepths + leftAllNodeDepths + rightAllNodeDepths

    return NodeInfo(nodeCount, nodeDepths, allNodeDepths)
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/compareLeafTraversal/solution1/compareLeafTraversal.kt

```
package ae.veryHard.compareLeafTraversal.solution1

import java.util.*

open class BinaryTree(value: Int) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
}

fun compareLeafTraversal(tree1: BinaryTree, tree2: BinaryTree): Boolean {
    val tree1Stack = Stack<BinaryTree>()
    val tree2Stack = Stack<BinaryTree>()

    tree1Stack.push(tree1)
    tree2Stack.push(tree2)

    while (tree1Stack.isNotEmpty() && tree2Stack.isNotEmpty()) {
        val leftNode1 = getNextLeafNode(tree1Stack)
        val leafNode2 = getNextLeafNode(tree2Stack)

        if (leftNode1.value != leafNode2.value) {
            return false
        }
    }

    return tree1Stack.isEmpty() && tree2Stack.isEmpty()
}

fun getNextLeafNode(stack: Stack<BinaryTree>): BinaryTree {
    var currentNode = stack.pop()

    while (!isLeafNode(currentNode)) {

        currentNode.left?.let { leftChild ->
            stack.push(leftChild)
        }

        currentNode.right?.let { rightChild ->
            stack.push(rightChild)
        }

        currentNode = stack.pop()
    }

    return currentNode
}

fun isLeafNode(node: BinaryTree): Boolean = node.left == null && node.right == null
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/linkedListPalindrome/solution1/linkedListPalindrome.kt

```
package ae.veryHard.linkedListPalindrome.solution1

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun linkedListPalindrome(head: LinkedList?): Boolean {
    var slow = head
    var fast = head

    // When fast itself is past last element (null)
    // Or it is the last element (it's next is null)
    // Then break the loop
    // fast != null && fast.next != null
    while (fast?.next != null) {
        slow = slow!!.next
        fast = fast.next!!.next
    }

    var current = head
    var revCurrent = reverseLinkedList(slow)

    // current can have more element
    // In other word for odd length palindromes
    // current can have one more element than revCurrent.
    // That one more element will have no pair
    while (revCurrent != null) {
        if (current!!.value != revCurrent.value) {
            return false
        }

        current = current.next
        revCurrent = revCurrent.next
    }

    return true
}

fun reverseLinkedList(node: LinkedList?): LinkedList? {
    var prev: LinkedList? = null
    var current = node

    while (current != null) {
        val next = current.next
        current.next = prev
        prev = current
        current = next
    }

    return prev
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/linkedListPalindrome/solution2/linkedListPalindrome.kt

```
package ae.veryHard.linkedListPalindrome.solution2

open class LinkedList(value: Int) {
    var value = value
    var next: LinkedList? = null
}

fun linkedListPalindrome(head: LinkedList?): Boolean {
    return isPalindrome(head, head).outerNodesAreEqual
}

fun isPalindrome(leftNode: LinkedList?, rightNode: LinkedList?):
LinkedListInfo {
    rightNode ?: return LinkedListInfo(true, leftNode)

    val (nextOuterCallsAreEqual, leftNodeToCompare) = isPalindrome(leftNode,
        rightNode.next)

    val outerNodesAreEqual = nextOuterCallsAreEqual && leftNodeToCompare!!.value == rightNode.value

    return LinkedListInfo(outerNodesAreEqual, leftNodeToCompare?.next)
}

data class LinkedListInfo(val outerNodesAreEqual: Boolean, val
leftNodeToCompare: LinkedList?)
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/minimumAreaRectangle/solution1/minimumAreaRectangle.kt

```
package ae.veryHard.minimumAreaRectangle.solution1

import kotlin.math.*

fun minimumAreaRectangle(points: List<List<Int>>): Int {
    val pointsSet = getPointsSet(points)

    var minArea = Integer.MAX_VALUE

    for (i in 0 .. points.lastIndex) {

        val p1 = points[i]
        val (p1X, p1Y) = p1

        for (j in i + 1 .. points.lastIndex) {

            val p2 = points[j]
            val (p2X, p2Y) = p2

            val areOnSameAxes = p1X == p2X || p1Y == p2Y

            if (areOnSameAxes) continue

            val diagonalKey1 = convertPointToString(listOf(p1X, p2Y))
            val diagonalKey2 = convertPointToString(listOf(p2X, p1Y))

            if (pointsSet.contains(diagonalKey1) && pointsSet.contains(
diagonalKey2)) {

                minArea = min(minArea, calculateArea(p1, p2))
            }
        }
    }

    return if (minArea == Integer.MAX_VALUE) 0 else minArea
}

fun getPointsSet(points: List<List<Int>>): Set<String> {
    val pointsSet = mutableSetOf<String>()

    points.forEach { point ->
        pointsSet.add(convertPointToString(point))
    }

    return pointsSet
}

fun convertPointToString(point: List<Int>): String {
    return point.joinToString("-")
}

fun calculateArea(point1: List<Int>, point2: List<Int>): Int {
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/minimumAreaRectangle/solution1/minimumAreaRectangle.kt

```
    return abs(point1[0] - point2[0]) * abs(point1[1] - point2[1])
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/minimumAreaRectangle/solution2/minimumAreaRectangle.kt

```
package ae.veryHard.minimumAreaRectangle.solution2

import kotlin.math.min

fun minimumAreaRectangle(points: List<List<Int>>): Int {

    // Return value is a SortedMap. So it's sorted by key
    // Which means that X values are distributed low to high (bottom to top)
}

val columnsMap = getColumnsMap(points)
val edgesParallelToXAxis = mutableMapOf<String, Int>()

var minArea = Integer.MAX_VALUE

columnsMap.map { (x1, yList) ->

    // Iterate through sorted column values (left to right)
    for (i in 0 .. yList.lastIndex) {

        val y1 = yList[i]

        // Now iterate through all of the y positions on the LEFT of
        // the current position
        // Implied by j to 0 until i
        for (j in 0 until i) {

            val y2 = yList[j]

            // Form a key with p1Y-p2Y
            // y1 is on the right and y2 is on the left (Example: 5-2)
            val parallelEdge = "$y1:$y2"

            // If in ANY OTHER row (X position) we have same y indexes
            // (like y1 and y2)
            if (edgesParallelToXAxis.containsKey(parallelEdge)) {

                val x2 = edgesParallelToXAxis[parallelEdge]!!

                // We have a rectangle parallel to X and Y axes
                // Calculate the area, remember y1 is greater than y2
                // And x1 is greater than the previous found x2 value
                val area = (y1 - y2) * (x1 - x2)
                minArea = min(minArea, area)
            }

            // Save x value against this "p1Y-p2Y" "computed" key
            // We only need the LAST x value. Because of the sorted map
            // Any previous values will result in rectangle with larger
            area
            // While we need minimum area rectangle.
            edgesParallelToXAxis[parallelEdge] = x1
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/minimumAreaRectangle/solution2/minimumAreaRectangle.kt

```
        }

    }

    return if (minArea == Integer.MAX_VALUE) 0 else minArea
}

/*
{ -1 : -2, }
{ 1 : 5, 2, }
{ 2 : 4, 2, 5, }
{ 4 : 2, 5, }
{ 5 : 1, }
*/

// Group all y indexes from same x index together
fun getColumnsMap(points: List<List<Int>>): Map<Int, List<Int>> {
    val columnsMap = sortedMapOf<Int, MutableList<Int>>()

    points.forEach { point ->

        val (pX, pY) = point

        if (!columnsMap.containsKey(pX)) {
            columnsMap[pX] = mutableListOf<Int>()
        }

        columnsMap[pX]!!.add(pY)
    }

    return columnsMap.mapValues { (_, yList) ->
        // Sort y positions for current x
        // After the sorting y positions will be distributed left to right
        yList.sorted()
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/twoEdgeConnectedGraph/solution1/twoEdgeConnectedGraph.kt

```
package ae.veryHard.twoEdgeConnectedGraph.solution1

import kotlin.math.min

fun twoEdgeConnectedGraph(edges: List<List<Int>>): Boolean {
    // Empty graph is two edge connected
    if (edges.isEmpty()) return true

    // Vector which keeps track of the arrival time
    val arrivalTimes = MutableList<Int>(edges.size) { -1 }

    // If back to the beginning of the dfs we have a -1 bubbled up, then
    return false
    if (getMinimumArrivalTimeToAncestor(0, -1, 0, arrivalTimes, edges) == -1) {
        return false
    }

    // If all of the edges were visited in the process then return true.
    // Else the graph has detached nodes and not even connected.
    return ifAllEdgesVisited(arrivalTimes)
}

fun getMinimumArrivalTimeToAncestor(currentNode: Int, parentNode: Int,
currentTime: Int, arrivalTimes: MutableList<Int>, edges: List<List<Int>>): Int {
    // currentNode is being visited at currentTime
    arrivalTimes[currentNode] = currentTime

    // Assumption. Minimum arrival time for current node is the current
    // time. Because a node can certainly have a back edge to itself
    // Also we need to get back edge to any of the current node's ancestors
    //

    // That's why we set minimumArrivalTime to currentTime.
    var minimumArrivalTime = currentTime

    // Iterate to all of the edges from the current node
    for (destination in edges[currentNode]) {
        // If we didn't visit this node before. i.e. This is a tree edge
        if (arrivalTimes[destination] == -1) {
            minimumArrivalTime = min(
                minimumArrivalTime,
                getMinimumArrivalTimeToAncestor(destination, currentNode,
                currentTime + 1, arrivalTimes, edges)
            )
        }
        // If the node was visited before but it is not the parent of the
        // current node, i.e. this is a back edge
        // So we need the minimum between minumumArrivalTime of the back
        // edge and minumumArrivalTime of current node.
        else if (destination != parentNode) {
            // This means that consider back edge but ignore forward edge
        }
    }
}
```

```
File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/twoEdgeConnectedGraph/solution1/twoEdgeConnectedGraph.kt
    minimumArrivalTime = min(minimumArrivalTime, arrivalTimes[
destination])
}
}

// It didn't find any back edge throughout the dfs "to any of current
node's ancestor"
// The logic of this algorithm. If any node stays in it's current
arrival time after the dfs is done and it's not the starting node, then it
has a bridge.
// And as a result the graph is not two edge connected. The -1 value
will be bubbled up throughout the recursion, because min(-1,
any_positive_value) is -1.
if (minimumArrivalTime == currentTime && parentNode != -1) {
    return -1
}

return minimumArrivalTime
}

fun ifAllEdgesVisited(arrivalTimes: List<Int>): Boolean {
    return arrivalTimes.find { it == -1 } == null
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/longestBalancedSubstring/solution1/longestBalancedSubstring.kt

```
package ae.veryHard.longestBalancedSubstring.solution1

import java.util.Stack
import kotlin.math.max

fun longestBalancedSubstring(string: String): Int {
    var maxLength = 0

    for (i in string.indices) {
        // Valid balanced substring should always have even length
        // Starting from length 2 "()" is sufficient.
        for (j in i + 2 .. string.length step 2) {
            if (isValid(string.substring(i, j))) {
                maxLength = max(maxLength, j - i)
            }
        }
    }
    return maxLength
}

fun isValid(string: String): Boolean {
    val stack = Stack<Char>()

    for (c in string) {
        if (c == '(') {
            stack.push(c)
        } else if (stack.isEmpty()) {
            return false
        } else {
            stack.pop()
        }
    }

    return stack.isEmpty()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/longestBalancedSubstring/solution2/longestBalancedSubstring.kt

```
package ae.veryHard.longestBalancedSubstring.solution2

import kotlin.math.max

fun longestBalancedSubstring(string: String): Int {
    // We need this for the outer balanced substring
    // length calculation, such as "((()) or ()"
    // This extra value will indicate after which index
    // the balanced substring will start.
    val idxStack = mutableListOf(-1)
    var maxLength = 0

    for (i in string.indices) {
        if (string[i] == '(') {
            idxStack.add(i)
        } else {
            idxStack.last()
            idxStack.removeAt(idxStack.lastIndex)

            if (idxStack.isNotEmpty()) {
                val balancedSubstringStartingIdx = idxStack.last()
                val currentLen = i - balancedSubstringStartingIdx
                maxLength = max(maxLength, currentLen)
            } else {
                // This added index indicates to any NEXT balanced
                // substring
                // That it starts after this index i, where the "unbalanced"
                //
                // substring ended.
                idxStack.add(i)
            }
        }
    }

    return maxLength
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/longestBalancedSubstring/solution3/longestBalancedSubstring.kt

```
package ae.veryHard.longestBalancedSubstring.solution3

import kotlin.math.max

fun longestBalancedSubstring(string: String): Int {
    return max(
        getLongestBalancedInDirection(string, true),
        getLongestBalancedInDirection(string, false)
    )
}

fun getLongestBalancedInDirection(string: String, isLeftToRight: Boolean): Int {
    val openingParen = if (isLeftToRight) '(' else ')'

    var maxLen = 0
    var openingCount = 0
    var closingCount = 0

    val range = if (isLeftToRight) {
        string.indices
    } else {
        string.indices.reversed()
    }

    for (i in range) {
        val currentChar = string[i]

        if (currentChar == openingParen) {
            openingCount += 1
        } else {
            closingCount += 1
        }

        if (openingCount == closingCount) {
            maxLen = max(maxLen, openingCount * 2)
        } else if (closingCount > openingCount) {
            openingCount = 0
            closingCount = 0
        }
    }
}

return maxLen
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/iterativeInOrderTraversal/solution1/iterativeInOrderTraversal.kt

```
package ae.veryHard.iterativeInOrderTraversal.solution1

open class BinaryTree(value: Int, parent: BinaryTree?) {
    var value = value
    var left: BinaryTree? = null
    var right: BinaryTree? = null
    var parent: BinaryTree? = parent
}

fun iterativeInOrderTraversal(tree: BinaryTree?, callback: (BinaryTree?) -> Unit) {
    var currentNode = tree
    var prevNode: BinaryTree? = null

    while (currentNode != null) {

        val nextNode = when(prevNode) {
            null, currentNode.parent -> {
                if (currentNode.left != null) {
                    currentNode.left
                } else {
                    callback(currentNode)
                    if (currentNode.right != null) {
                        currentNode.right
                    } else {
                        currentNode.parent
                    }
                }
            }
            currentNode.left -> {
                callback(currentNode)
                if (currentNode.right != null) {
                    currentNode.right
                } else {
                    currentNode.parent
                }
            }
            else -> {
                currentNode.parent
            }
        }

        prevNode = currentNode
        currentNode = nextNode
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/knuthMorrisPrattAlgorithm/solution1/knuthMorrisPrattAlgorithm.kt

```
package ae.veryHard.knuthMorrisPrattAlgorithm.solution1

// O(n + m) Time | O(m) Space
fun knuthMorrisPrattAlgorithm(string: String, substring: String): Boolean {
    val pattern = buildPattern(substring)
    return doesMatch(string, substring, pattern)
}

// Uses repeated patterns in string to perform string matching
// https://imgur.com/a/0deXXFE
fun buildPattern(str: String): List<Int> {
    val pattern = mutableListOf(str.length) { -1 }

    var i = 1
    var j = 0

    while (i < str.length) {
        if (str[i] == str[j]) {
            pattern[i] = j
            i += 1
            j += 1
        } else if (j > 0) {
            j = pattern[j - 1] + 1
        } else {
            i += 1
        }
    }

    return pattern
}

fun doesMatch(str: String, substr: String, pattern: List<Int>): Boolean {
    var i = 0
    var j = 0

    // Until there are enough characters to match substr
    while (i + substr.length - j <= str.length) {
        if (str[i] == substr[j]) {

            if (j == substr.lastIndex) {
                return true
            }

            i += 1
            j += 1
        } else if (j > 0) {
            j = pattern[j - 1] + 1
        } else {
            i += 1
        }
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/knuthMorrisPrattAlgorithm/solution1/knuthMorrisPrattAlgorithm.kt

```
    return false  
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/maxProfitWithKTransactions/solution1/maxProfitWithKTransactions.kt

```
package ae.veryHard.maxProfitWithKTransactions.solution1

import kotlin.math.max

fun maxProfitWithKTransactions(prices: List<Int>, k: Int): Int {
    if (prices.isEmpty()) return 0

    val profits = MutableList(k + 1) {
        MutableList(prices.size) { 0 }
    }

    for (t in 1 .. k) {

        var maxSoFar = Integer.MIN_VALUE

        for (d in 1 .. prices.lastIndex) {

            maxSoFar = max(maxSoFar, profits[t - 1][d - 1] - prices[d - 1])
            profits[t][d] = max(profits[t][d - 1], prices[d] + maxSoFar)
        }
    }

    return profits[k][prices.lastIndex]
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/smallestSubStringContaining/solution1/smallestSubStringContaining.kt

```
package ae.veryHard.smallestSubStringContaining.solution1

fun smallestSubStringContaining(bigString: String, smallString: String): String {
    val smallStringCounts = getCharacterCounts(smallString)
    val smallestSubStringBounds = getSmallestSubstrBounds(bigString,
    smallStringCounts)
    return getSubstringFromBounds(bigString, smallestSubStringBounds)
}

fun getCharacterCounts(string: String): Map<Char, Int> {
    return string.fold(mapOf()) { acc, c ->
        if (acc.containsKey(c)) {
            acc.plus(c to acc[c]!! + 1)
        } else {
            acc.plus(c to 1)
        }
    }
}

fun getSmallestSubstrBounds(bigStr: String, targetCharCounts: Map<Char, Int>): Pair<Int, Int> {
    // Initial bound. Biggest possible bound
    var substrBounds = mutableListOf(0, Integer.MAX_VALUE)
    val targetUniqueCharCount = targetCharCounts.size
    var currentUniqueCharCount = 0

    val currentCharCounts = mutableMapOf<Char, Int>()
    var leftIdx = 0
    var rightIdx = 0

    while (rightIdx < bigStr.length) {
        val rightChar = bigStr[rightIdx]

        // If target map doesn't contain this char, then ignore and move
        forward
        if (!targetCharCounts.containsKey(rightChar)) {
            rightIdx += 1
            continue
        }

        increaseCharacterCount(currentCharCounts, rightChar)
        if (currentCharCounts[rightChar] == targetCharCounts[rightChar]) {
            currentUniqueCharCount += 1
        }

        while (currentUniqueCharCount == targetUniqueCharCount && leftIdx
        <= rightIdx) {
            // Gap between 0 and INT_MAX is the biggest possible gap. So
            any substring will be less than that length
            substrBounds = getSmallerSubstrBounds(substrBounds[0],
            substrBounds[1], leftIdx, rightIdx)
        }
    }
}
```

```

    val leftChar = bigStr[leftIdx]

        // If target map doesn't contain this char, then this is
        // useless character. ignore and move forward
        if (!targetCharCounts.containsKey(leftChar)) {
            leftIdx += 1
            continue
        }

        // aaa is less than aaaa. But we can't decrement
        currentUniqueCharCount each time we go down from aaa to aa to a etc.
        // This could happen if we did if (currentCharCounts[leftChar]
        ] < targetCharCounts[leftChar])
        // If we lost an essential character while sliding leftIdx then
        decrement the unique character count.
        if (currentCharCounts[leftChar] == targetCharCounts[leftChar]
    ) {
        currentUniqueCharCount -= 1
    }

        // And for the above reason we decrease after checking current
        charCounts and decrementing currentUniqueCharCount based on that
        decreaseCharacterCount(currentCharCounts, leftChar)

        leftIdx += 1
    }

    rightIdx += 1
}

return substrBounds[0] to substrBounds[1]
}

fun getSubstringFromBounds(string: String, bounds: Pair<Int, Int>): String
{
    if (bounds.second == Integer.MAX_VALUE) return ""
    return string.substring(bounds.first .. bounds.second)
}

fun increaseCharacterCount(charCounts: MutableMap<Char, Int>, char: Char) {
    if (charCounts.containsKey(char)) {
        charCounts[char] = charCounts[char]!! + 1
    } else {
        charCounts[char] = 1
    }
}

fun decreaseCharacterCount(charCounts: MutableMap<Char, Int>, char: Char) {
    charCounts[char] = charCounts[char]!! - 1
}

```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/smallestSubstringContaining/solution1/smallestSubstringContaining.kt

```
fun getSmallerSubstrBounds(l1: Int, r1: Int, l2: Int, r2: Int): MutableList<Int> {
    return if (r1 - l1 < r2 - l2) {
        mutableListOf(l1, r1)
    } else {
        mutableListOf(l2, r2)
    }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/smallestSubstringContaining/solution2/smallestSubstringContaining.kt

```
package ae.veryHard.smallestSubstringContaining.solution2

fun smallestSubstringContaining(bigString: String, smallString: String): String {
    val tMap = mutableMapOf<Char, Int>()
    for (c in smallString) {
        if (!tMap.containsKey(c)) {
            tMap[c] = 0
        }
        tMap[c] = tMap[c]!! + 1
    }

    val sMap = mutableMapOf<Char, Int>()
    var formed = 0

    var left = 0
    var right = 0
    var result = mutableListOf(0, Integer.MAX_VALUE)

    while (right < bigString.length) {

        val c = bigString[right]

        if (!sMap.containsKey(c)) {
            sMap[c] = 0
        }
        sMap[c] = sMap[c]!! + 1

        if (tMap.containsKey(c) && sMap[c]!! == tMap[c]!!) {
            formed += 1
        }

        while (left <= right && formed == tMap.size) {
            if (right - left < result[1] - result[0]) {
                result = mutableListOf(left, right)
            }

            val leftChar = bigString[left]

            if (sMap[leftChar] == tMap[leftChar]) {
                formed -= 1
            }
            sMap[leftChar] = sMap[leftChar]!! - 1

            left += 1
        }

        right += 1
    }

    return if (result[1] == Integer.MAX_VALUE) "" else bigString.substring(
        result[0] .. result[1])
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/smallestSubstringContaining/solution2/smallestSubstringContaining.kt
}

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/longestIncreasingSubsequence/solution1/longestIncreasingSubsequence.kt

```
package ae.veryHard.longestIncreasingSubsequence.solution1

fun longestIncreasingSubsequence(array: List<Int>): List<Int> {
    val lengths = MutableList<Int>(array.size) { 1 }
    val sequences = MutableList<Int>(array.size) { -1 }

    var longestIdx = 0

    for (i in 0 .. array.lastIndex) {

        for (j in 0 until i) {

            if (array[j] < array[i] && lengths[j] + 1 > lengths[i]) {
                lengths[i] = lengths[j] + 1
                sequences[i] = j
            }
        }

        if (lengths[i] > lengths[longestIdx]) {
            longestIdx = i
        }
    }

    return buildSequence(sequences, array, longestIdx)
}

fun buildSequence(sequences: List<Int>, array: List<Int>, longestIdx: Int): List<Int> {
    var currentIdx = longestIdx
    val result = mutableListOf<Int>()

    while (currentIdx != -1) {
        result.add(array[currentIdx])
        currentIdx = sequences[currentIdx]
    }

    // Or result.reverse() and return result
    return result.reversed()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/numberOfBinaryTreeTopologies/solution1/numberOfBinaryTreeTopologies.kt

```
package ae.veryHard.numberOfBinaryTreeTopologies.solution1

// O((n*(2n)!)/(n!(n + 1)!)) Time | O(n) space
fun numberOfBinaryTreeTopologies(n: Int): Int {
    if (n == 0) return 1

    var topologies = 0

    for (l in 0 until n) {
        val r = n - 1 - l // 1 less for the root
        val fl = numberOfBinaryTreeTopologies(l)
        val fr = numberOfBinaryTreeTopologies(r)

        topologies += (fl * fr)
    }

    return topologies
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/numberOfBinaryTreeTopologies/solution2/numberOfBinaryTreeTopologies.kt

```
package ae.veryHard.numberOfBinaryTreeTopologies.solution2

// O(n ^ 2) Time | O(n) Space
fun numberOfBinaryTreeTopologies(n: Int): Int {
    val memo = mutableMapOf<Int, Int>(0 to 1)
    return recurseHelper(n, memo)
}

fun recurseHelper(n: Int, memo: MutableMap<Int, Int>): Int {
    if (memo.containsKey(n)) {
        return memo[n]!!
    }

    var topologies = 0
    for (l in 0 until n) {
        val r = n - 1 - l
        val fl = recurseHelper(l, memo)
        val fr = recurseHelper(r, memo)
        topologies += (fl * fr)
    }
    memo[n] = topologies

    return memo[n]!!
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/numberOfBinaryTreeTopologies/solution3/numberOfBinaryTreeTopologies.kt

```
package ae.veryHard.numberOfBinaryTreeTopologies.solution3

// O(n ^ 2) Time | O(n) Space
fun numberOfBinaryTreeTopologies(n: Int): Int {
    val memo = mutableListOf<Int>(1)

    for (m in 1 .. n) {

        var numTopologies = 0

        for (l in 0 until m) {
            val r = m - 1 - l
            val fl = memo[l]
            val fr = memo[r]
            numTopologies += (fl * fr)
        }

        memo.add(numTopologies)
    }

    return memo.last()
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/palindromePartitioningMinCuts/solution1/palindromePartitioningMinCuts.kt

```
package ae.veryHard.palindromePartitioningMinCuts.solution1

fun palindromePartitioningMinCuts(str: String): Int {
    val cuts = mutableListOf(str.length) { 0 }

    for (i in 0 .. str.lastIndex) {
        if (isPalindrome(str, 0, i)) {
            cuts[i] = 0
        } else {
            cuts[i] = cuts[i - 1] + 1

            for (j in 1 until i) {
                // If str[j] to str[i] is a palindrome
                // And if we add a cut here and it becomes smaller than
                // current cut at i.
                if (isPalindrome(str, j, i) && cuts[j - 1] + 1 < cuts[i]) {
                    cuts[i] = cuts[j - 1] + 1
                }
            }
        }
    }

    return cuts.last()
}

fun isPalindrome(str: String, i: Int, j: Int): Boolean {
    var idx1 = i
    var idx2 = j

    while (idx1 < idx2) {
        if (str[idx1] != str[idx2]) {
            return false
        }

        idx1++
        idx2--
    }

    return true
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/palindromePartitioningMinCuts/solution2/palindromePartitioningMinCuts.kt

```
package ae.veryHard.palindromePartitioningMinCuts.solution2

fun palindromePartitioningMinCuts(str: String): Int {
    val palindromesArray = preparePalindromesArray(str)

    val cuts = mutableListOf(str.length) { 0 }

    for (i in 0 .. str.lastIndex) {
        if (palindromesArray[0][i]) {
            cuts[i] = 0
        } else {
            cuts[i] = cuts[i - 1] + 1

            for (j in 1 until i) {
                // If str[j] to str[i] is a palindrome
                // And if we add a cut here and it becomes smaller than
                // current cut at i.
                if (palindromesArray[j][i] && cuts[j - 1] + 1 < cuts[i]) {
                    cuts[i] = cuts[j - 1] + 1
                }
            }
        }
    }

    return cuts.last()
}

fun preparePalindromesArray(str: String): List<List<Boolean>> {
    val palindromes = mutableListOf(str.length) {
        mutableListOf(str.length) { false }
    }

    for (i in 0 .. str.lastIndex) {
        palindromes[i][i] = true
    }

    for (length in 2 .. str.length) {
        for (i in 0 .. str.length - length) {

            val j = i + length - 1

            if (length == 2) {
                palindromes[i][j] = str[i] == str[j]
            } else {
                palindromes[i][j] = str[i] == str[j] && palindromes[i + 1][
j - 1]
            }
        }
    }

    return palindromes.map { it.toList() }
}
```

File - /home/ashfaq/IdeaProjects/practice/src/main/kotlin/ae/veryHard/palindromePartitioningMinCuts/solution2/palindromePartitioningMinCuts.kt
}