Algorithms in Kotlin

Jaideep Ganguly, Sc.D.

Contents

CHAPTER 1

STACK, QUEUE, LINKED LIST

1.1 Stack

Listing 1.1 – Stack.

```
package dsa
import java.util.*

public class DST_Stack<T>: Collection<T> {

    public          var   head: Node<T>? = null
    public override var   size: Int      = 0

    public class Node<T>(var value: T) {
        var next: Node<T>? = null
    }

    public fun push(item: T) {
        val node = Node(item)
        node.next = head
        head = node
        size++
    }

    public fun pop()  {
        head = head!!.next
    }

    public fun peek(): T {
        if (size == 0) throw NoSuchElementException()
        return head!!.value
    }

    public override fun isEmpty(): Boolean {
        return size == 0
    }

    public override fun containsAll(elements: Collection<T>): Boolean {
        for (element in elements) {
            if (!contains(element)) return false
        }
        return true
    }

    public override fun iterator(): Iterator<T> {
        return object: Iterator<T> {
            var node = head

            override fun hasNext(): Boolean {
                return node != null
            }

            override fun next(): T {
                if (!hasNext()) throw NoSuchElementException()

                val current = node!!
                node = current.next
                return current.value
            }
        }
    }
}
```

1.2 Queue

Listing 1.2 – Stack.

```
package dsa
import java.util.*

public class DST_Queue<T> : Collection<T> {

    public          var   head: Node<T>? = null
    public          var   tail: Node<T>? = null
    public override var   size: Int      = 0

    public class Node<T>(var value: T) {
        var next: Node<T>? = null
    }

    public fun enqueu(item: T) {

        val node = Node(item)
        val tail = this.tail

        if (tail == null) {
            this.head = node
            this.tail = node
        } else {
            tail.next = node
            this.tail = node
        }
        size++
    }

    public fun dequeue()  {
        if (head == null)
            return
        head = head?.next
    }

    public fun peek(): T {
        if (size == 0) throw NoSuchElementException()
        return head!!.value
    }

    public override fun isEmpty(): Boolean {
        return size == 0
    }

    public override fun contains(element: T): Boolean {
        for (obj in this) {
            if (obj == element) return true
        }
        return false
    }

    public override fun containsAll(elements: Collection<T>): Boolean {
        for (element in elements) {
            if (!contains(element)) return false
        }
        return true
    }

    public override fun iterator(): Iterator<T> {

        return object : Iterator<T> {
            var node = head

            override fun hasNext(): Boolean {
                return node != null
            }

            override fun next(): T {
                if (!hasNext()) throw NoSuchElementException()

                val current = node!!
                node = current.next
                return current.value
            }
        }
    }
}
```

1.3 Linked List

Listing 1.3 – Stack.

```
package dsa


class Node<T>(value: T){
    var value: T = value
    var next: Node<T>? = null
    var prev: Node<T>? = null
}

class LinkedList<T> {

    private var head:Node<T>? = null

    var isEmpty: Boolean = (head == null)

    fun first(): Node<T>? = head

    fun last(): Node<T>? {
        var node = head
        if (node != null){
            while (node?.next != null) {
                node = node?.next
            }
            return node
        } else {
            return null
        }
    }

    fun count(): Int {
        var node = head
        if (node != null){
            var counter = 1
            while (node?.next != null){
                node = node?.next
                counter += 1
            }
            return counter
        } else {
            return 0
        }
    }

    fun nodeAtIndex(index: Int) : Node<T>? {
        if (index >= 0) {
            var node = head
            var i = index
            while (node != null) {
                if (i == 0) return node
                i -= 1
                node = node.next
            }
        }
        return null
    }

    fun append(value: T) {
        var newNode = Node(value)

        var lastNode = this.last()
        if (lastNode != null) {
            newNode.prev = lastNode
            lastNode.next = newNode
        } else {
            head = newNode
        }
    }

    fun removeAll() {
        head = null
    }

    fun removeNode(node: Node<T>):T {
        val prev = node.prev
        val next = node.next

        if (prev != null) {
            prev.next = next
        } else {
            head = next
        }
        next?.prev = prev

        node.prev = null
        node.next = null

        return node.value
    }

    fun removeLast() : T? {
        val last = this.last()
        if (last != null) {
            return removeNode(last)
        } else {
            return null
        }
    }

    fun removeAtIndex(index: Int): T? {
        val node = nodeAtIndex(index)
        if (node != null) {
            return removeNode(node)
        } else {
            return null
        }
    }

    override fun toString(): String {
        var s = "["
        var node = head
        while (node != null) {
            s += "${node.value}"
            node = node.next
            if (node != null) { s += ", " }
        }
        return s + "]"
    }
}
```

CHAPTER 2

SEARCH

2.1 Linear Search

Listing 2.1 – Linear Search.

```
public class SEA_Linear {

    fun <T : Comparable<T>> linearSearch(list: List<T>, key: T): Int? {
        for ((index, value) in list.withIndex()) {
            if (value == key) return index
        }
        return null
    }
}
```

2.2 Binary Search (for Ordered Lists)

Listing 2.2 – Binary Search.

```
 1 public class SEA_Linear {
 2 class SEA_BinarySearch <T> {
 3     /* Comparable is an interface that is being extended by T
 4        The following will not work, will error in line 14
 5        fun binarySearch(list: List<T>, key: T): Int? {
 6     */
 7     fun <T: Comparable<in T>> binarySearch(list: List<T>, key: T): Int? {
 8         var rangeStart = 0
 9         var rangeEnd = list.count()
10         while (rangeStart < rangeEnd) {
11             val midIndex = rangeStart + (rangeEnd - rangeStart)/2
12             if (list[midIndex] == key) {
13                 return midIndex
14             } else if (list[midIndex] < key) {
15                 rangeStart = midIndex + 1
16             } else {
17                 rangeEnd = midIndex
18             }
19         }
20         return null
21     }
22 }
```

CHAPTER 3

SORT

3.1 Bubble Sort

Listing 3.1 – Bubble Sort.

```
public fun bubblesort(inp: IntArray)  {
    var swappedElements : Boolean;
    do {
        swappedElements = false;
        for (i in 0..inp.size - 2){
            if (inp[i] > inp[i + 1]){
                val tmp = inp[i+1]
                inp[i+1] = inp[i]
                inp[i] = tmp
                swappedElements = true;
            }
        }
    } while (swappedElements);
}
```

3.2 Insertion Sort

Listing 3.2 – Insertion Sort.

```
fun insertionSort(inp: IntArray) {

    for (i in 1..(inp.size)-1) {
        val value = inp[i]
        var hole = i
        while ( (hole > 0) &&
                (inp[hole-1] > value)) {
            inp[hole] = inp[hole - 1]
            hole = hole - 1
        }
        inp[hole] = value
    }
}
```

3.3 Merge Sort

Listing 3.3 – Merge Sort.

```
fun mergeSort(list: List<Int>): List<Int> {
    if (list.size <= 1) {
        return list
    }

    val middle = list.size / 2
    var left = list.subList(0, middle);
    var right = list.subList(middle, list.size);

    return merge(mergeSort(left), mergeSort(right))
}

fun merge(left: List<Int>, right: List<Int>): List<Int> {
    var indexLeft = 0
    var indexRight = 0
    var newList: MutableList<Int> = mutableListOf()

    while (indexLeft < left.count() && indexRight < right.count()) {
        if (left[indexLeft] <= right[indexRight]) {
            newList.add(left[indexLeft])
            indexLeft++
        } else {
            newList.add(right[indexRight])
            indexRight++
        }
    }

    while (indexLeft < left.size) {
        newList.add(left[indexLeft])
        indexLeft++
    }

    while (indexRight < right.size) {
        newList.add(right[indexRight])
        indexRight++
    }

    return newList;
}
```

3.4 Qsort

Listing 3.4 – QSort.

```
 fun <T: Comparable<T>> quicksort(items: List<T>): List<T> {
    if (items.count() < 2) {
        return items
    }
    val pivot = items[items.count() / 2]
    val equal = items.filter { it == pivot }
    val less = items.filter { it < pivot }
    val greater = items.filter { it > pivot }
    return quicksort(less) + equal + quicksort(greater)
}
```

3.5 Selection Sort

Listing 3.5 – Selection Sort.

```kotlin
fun <T : Comparable<T>> selectionsort(items: MutableList<T>):
                            MutableList<T> {

    if (items.isEmpty()) {
        return items
    }

    for (idx in 0..items.count()) {
        val array = items.subList(0, items.count() - idx)
        val minItem = array.min()
        val indexOfMinItem = array.indexOf(minItem)

        if (minItem != null) {
            items.removeAt(indexOfMinItem)
            items.add(minItem)
        }
    }

    for (x in items)
        println("$x ")
    println()
    return items
}
```

3.6 Shell Sort

Listing 3.6 – Shell Sort.

```kotlin
fun shellSort(inp: IntArray) {
    var gap = 1
    var value: Int
    var index: Int

    while (gap < inp.size / 3) {
        gap = 3 * gap + 1
    }

    while (gap > 0) {

        for (i in gap until inp.size step 1) {
            value = inp[i]
            index = i

            while ( (index >= gap) &&
                    (value < inp[index - gap]) ) {

                inp[index] = inp[index - gap]
                index = index - gap
                readLine()
            }

            inp[index] = value
        }

        gap = (gap - 1) / 3
    }

    for (x in inp)
        print("$x ")
    println()
}
```

CHAPTER 4

BINARY TREE

4.1 Depth First Search

Listing 4.1 – Depth First Search.

```
package dsa

import java.util.*

public class N(var value: Int, var left: N?, var right: N?)

class TRE_Traverse {

    fun preorder(root: N?) {
        if (root == null)
            return
        print("${root.value} ")
        if (root.left != null)
            preorder(root!!.left)
        if (root.right != null)
            preorder(root!!.right)
    }

    fun inorder(root: N?) {
        if (root == null)
            return
        if (root.left != null)
            inorder(root.left)
        print("${root.value} ")
        if (root.right != null)
            inorder(root.right)
    }

    fun postorder(root: N?) {
        if (root == null)
            return
        if (root.left != null)
            postorder(root.left)
        if (root.right != null)
            postorder(root.right)
        print("${root.value} ")
    }
```

4.2 Breadth First Search

Listing 4.2 – Breadth First Search.

```
fun bfs(root: N) {
    val q: Queue<N> = ArrayDeque<N>()

    if (root == null)
        return;

    q.add(root)
    while (!q.isEmpty()) {
        val current = q.remove()
        print("${current.value} ")
        if (current.left != null)
            q.add(current.left)
        if (current.right != null)
            q.add(current.right)
    }
    println()
}
```

4.3 IS Binary Search Tree

Listing 4.3 – Is BST.

```
fun isBST(root: N) : Boolean {
    if (isSubTreeLesser(root.left!!, root.value) &&
            isSubTreeGreater(root.right!!, root.value) &&
            isBST(root.left!!) &&
            isBST(root.right!!)) {
        return(true)
    }
    else
        return(false)
}

fun isSubTreeLesser(root: N, value: Int) : Boolean {
    if (root == null)
        return(true)

    if ( (root.value < value) &&
                    isSubTreeLesser(root.left!!,value) &&
                    isSubTreeGreater(root.right!!,value)) {
        return(true)
    }
    else
        return(false)
}

fun isSubTreeGreater(root: N, value: Int) : Boolean {
    if (root == null)
        return(true)
    if ( (root.value > value) &&
            isSubTreeLesser(root.left!!,value) &&
            isSubTreeGreater(root.right!!,value)) {
        return(true)
    }
    else
        return(false)
}
```

4.4 Insert, Delete Node

Listing 4.4 – Insert, Delete Node.

```
fun insertNode(root: N?, key: Int): N {
        var root = root

        if (root == null) { // If the tree is empty, return a new node
            return root!!
        }

        /* Otherwise, recur down the tree */
        if (key < root.value)
            root.left = insertNode(root.left!!, key)
        else if (key > root.value)
            root.right = insertNode(root.right!!, key)

        return root!!   // return the (unchanged) node pointer
    }

// A recursive function to insert a new key in BST
fun deleteNode(root: N?, key: Int): N? {

    // If the tree is empty, return
    if (root == null)
        return root

    // Otherwise, recur down the tree
    if (key < root.value)
        root.left = deleteNode(root.left, key)

    else if (key > root.value)
        root.right = deleteNode(root.right, key)

    else {  // if key is same as root's key, then delete node
        // node with only one child or no child
        if (root.left == null)
            return root.right

        else if (root.right == null)
            return root.left

        // node with 2 children;
        // Get inorder successor (smallest in right subtree)
        root.value = minValue(root.right)

        // Delete the inorder successor
        root.right = deleteNode(root.right, root.value)
    }

    return root
}

fun minValue(root: N?): Int {

    var root = root
    var minv = root!!.value

    while (root?.left!! != null) {
        minv = root?.left!!.value
        root = root?.left!!
    }
    return minv
}
```

4.5 Sorted List to BST

Listing 4.5 – Sorted List to BST.

```
fun sortedArrayToBST(arr: IntArray, start: Int, end: Int): Node? {

    if (start > end) {
        return null
    }

    // Get the middle element and make it root
    val mid = (start + end) / 2
    val node = Node(arr[mid])

    // Recursively construct left subtree; make it left child of root
    node.left = sortedArrayToBST(arr, start, mid - 1)

    // Recursively construct right subtree; make it right child of root
    node.right = sortedArrayToBST(arr, mid + 1, end)

    return node
}
```

CHAPTER 5

MINIMUM SPANNING TREE

5.1 Kruskal’S Algorithm

```
package kruskal

public class MSP_Kruskal {
    var mlSeg = mutableListOf<Seg>()

    init {

        mlSeg.add(Seg(0,0,1,10))
        mlSeg.add(Seg(1,1,2, 8))
        mlSeg.add(Seg(2,2,3, 7))
        mlSeg.add(Seg(3,3,4, 9))
        mlSeg.add(Seg(4,4,5,10))
        mlSeg.add(Seg(5,5,6, 2))
        mlSeg.add(Seg(6,6,7, 1))
        mlSeg.add(Seg(7,7,0, 8))
    }

    fun kruskal() {
        var msp = mutableListOf<Seg>()

        mlSeg.sortBy { it.ds }

        var mspSize = 0
        while (true) {

            var iterator = mlSeg.listIterator()     // To modify collection

            while (iterator.hasNext()) {
                var seg = iterator.next()

                if (!(isCycle(msp, seg))) {
                    iterator.remove()
                    msp.add(seg)
                    mspSize++
                }
            }

            if (msp.size == mspSize)
                break
        }

        printColl(msp)
    }

    fun isCycle(msp: List<Seg>, seg: Seg): Boolean {

        val con1 = msp.filter { (it.n1 == seg.n1) || (it.n2 == seg.n1) }.
                count()
        val con2 = msp.filter { (it.n1 == seg.n2) || (it.n2 == seg.n2) }.
                count()

        if (( con1 == 0) && (con2 == 0) ) {
            return false
        }
        else if ( ( (con1 != 0) && (con2 == 0) ) ||
                  ( (con1 == 0) && (con2 != 0) ) ) {
            return false
        }
        else {
            return true
        }

    }

}

data class Seg (val id: Int, val n1: Int, val n2: Int, val ds: Int)

fun printColl(coll: List<Any>) {
    for (c in coll)
        println(c)
    println()
}
```

CHAPTER 6

GRAPH TRAVERSAL

6.1 Djikstra's Algorithm

The algoritm requires tracking **visited** and **unvisited** nodes in two lists. Note the cost table:

Node	Cost	From
A	0	
B	10	C
C	∞	
...	...	...

6.2 Code

Listing 6.1 – Code.

```
package djikstra

public class Graph {
    var mlVisited = mutableListOf<Int>()
    var mlUnvisited = mutableListOf<Int>()

    var mlConn = mutableListOf<List<Int>>()

    var mlCost = mutableListOf<Cost>()

    init {
        mlUnvisited.add(0)
        mlUnvisited.add(1)
        mlUnvisited.add(2)
        mlUnvisited.add(3)
        mlUnvisited.add(4)

        mlConn.add(mutableListOf( 0, 6, -1,  1, -1))
        mlConn.add(mutableListOf( 6, 0,  5,  2,  2))
        mlConn.add(mutableListOf(-1, 5,  0, -1,  5))
        mlConn.add(mutableListOf( 1, 2, -1,  0,  1))
        mlConn.add(mutableListOf(-1, 2,  5,  1,  0))

        mlCost.add(Cost(0, 0, 0))
        mlCost.add(Cost(1, Int.MAX_VALUE, -1))
        mlCost.add(Cost(2, Int.MAX_VALUE, -1))
        mlCost.add(Cost(3, Int.MAX_VALUE, -1))
        mlCost.add(Cost(4, Int.MAX_VALUE, -1))
    }

    fun traverse(startFrom: Int) {
        var from = startFrom

        while (mlUnvisited.isNotEmpty()) {
            var listOfNodeDistPair = getListOfNodeDistPair(from)
            var listOfNodeDistPairUnvisited =
                    listOfNodeDistPair.filter { (n,d) -> !(n in mlVisited) }
            if ( listOfNodeDistPairUnvisited.isNotEmpty() ) {
                updateCost(from, listOfNodeDistPairUnvisited)
                mlVisited.add(from)
                mlUnvisited.remove(from)
                from = listOfNodeDistPairUnvisited.get(0).first
            }
            else
                break
        }
        for (c in mlCost)
            println(c)
    }

    // Get Node and Distance, sorted by distance
    fun getListOfNodeDistPair(from: Int): List<Pair<Int,Int>> {

        var mlND = mutableListOf<Pair<Int,Int>>()
        val row = mlConn.get(from)

        for (i in 0..row.size-1) {
            val d = row.get(i)
            if ((d != 0) && (d != -1)) {
                mlND.add(Pair(i, d))
            }
        }
        mlND.sortBy { (x,y) -> y }

        return mlND
    }

    // Update Cost
    fun updateCost(from: Int, listNodeDistPair: List<Pair<Int,Int>>) {

        val costUptoFrom = mlCost.get(from).dist

        for (p in listNodeDistPair) {

            val to = p.first
            val ds = p.second

            val currCost = mlCost.get(to).dist
            val newCost = costUptoFrom + ds

            if (newCost < currCost) {
                mlCost.get(to).dist = newCost
                mlCost.get(to).prev = from
            }

        }
    }
}


data class Conn(val n1: Int, val n2: Int, val dist: Int)
data class Cost(val node: Int, var dist: Int, var prev: Int)
```

CHAPTER 7

GREEDY

7.1 Knapsack

Listing 7.1 – Knapsack.

```kotlin
package dsa

class Knapsack {
    val wants = listOf(
            Item("map",        9, 150),
            Item("compass",  13,  35),
            Item("water",   153, 200),
            Item("sandwich", 50, 160),
            Item("glucose",  15,  60),
            Item("tin",      68,  45),
            Item("banana",   27,  60),
            Item("apple",    39,  40)
    )

    val MAX_WEIGHT = 400

    init {
        val (chosenItems, totalWeight, totalValue) =
                m(wants.size - 1, MAX_WEIGHT)
        println("Knapsack Item Chosen    Weight Value")
        for (item in chosenItems.sortedByDescending { it.value} )
            println("${item.name.padEnd(24)}  ${"%3d".format(item.weight)} ${"%3d".format(item.value)}")
        println("Total ${chosenItems.size} Items Chosen      $totalWeight $totalValue")
    }

    fun m(i: Int, w: Int): Triple<MutableList<Item>, Int, Int> {
        val chosen = mutableListOf<Item>()

        if (i < 0 || w == 0)
            return Triple(chosen, 0, 0)
        else if (wants[i].weight > w)
            return m(i - 1, w)

        val (l0, w0, v0) = m(i - 1, w)
        var (l1, w1, v1) = m(i - 1, w - wants[i].weight)

        v1 += wants[i].value
        if (v1 > v0) {
            l1.add(wants[i])
            return Triple(l1, w1 + wants[i].weight, v1)
        }

        return Triple(l0, w0, v0)
    }

    data class Item(val name: String, val weight: Int, val value: Int)
}
```