
ALGORITHMS IN KOTLIN

Jaideep Ganguly, Sc.D.



CONTENTS

Contents i

1	STACK, QUEUE, LINKED LIST	1
1.1	Stack	1
1.2	Queue	2
1.3	Linked List	3
2	SEARCH	7
2.1	Linear Search	7
2.2	Binary Search (for Ordered Lists)	7
3	SORT	9
3.1	Bubble Sort	9
3.2	Insertion Sort	9
3.3	Merge Sort	10
3.4	Qsort	10
3.5	Selection Sort	11
3.6	Shell Sort	11
4	BINARY TREE	13
4.1	Depth First Search	13
4.2	Breadth First Search	14
4.3	IS Binary Search Tree	14
4.4	Insert, Delete Node	15
4.5	Sorted List to BST	16
5	MINIMUM SPANNING TREE	17
5.1	Kruskal'S Algorithm	17
6	GRAPH TRAVERSAL	19
6.1	Dijkstra's Algorithm	19
6.2	Code	19
7	GREEDY	21
7.1	Knapsack	21
	Index	23

STACK, QUEUE, LINKED LIST

1.1 Stack

LISTING 1.1 – Stack.

```
1 package dsa
2 import java.util.*
3
4 public class DST_Stack<T>: Collection<T> {
5
6     public var head: Node<T>? = null
7     public override var size: Int = 0
8
9     public class Node<T>(var value: T) {
10         var next: Node<T>? = null
11     }
12
13     public fun push(item: T) {
14         val node = Node(item)
15         node.next = head
16         head = node
17         size++
18     }
19
20     public fun pop() {
21         head = head!!.next
22     }
23
24     public fun peek(): T {
25         if (size == 0) throw NoSuchElementException()
26         return head!!.value
27     }
28
29     public override fun isEmpty(): Boolean {
30         return size == 0
31     }
32
33     public override fun containsAll(elements: Collection<T>): Boolean {
34         for (element in elements) {
35             if (!contains(element)) return false
36         }
37         return true
38     }
39
40     public override fun iterator(): Iterator<T> {
41         return object: Iterator<T> {
42             var node = head
43
44             override fun hasNext(): Boolean {
45                 return node != null
46             }
47
48             override fun next(): T {
49                 if (!hasNext()) throw NoSuchElementException()
50             }
51         }
```

```

51         val current = node!!
52         node = current.next
53         return current.value
54     }
55 }
56 }
57 }

```

1.2 Queue

LISTING 1.2 – Stack.

```

1 package dsa
2 import java.util.*
3
4 public class DST_Queue<T> : Collection<T> {
5
6     public var head: Node<T>? = null
7     public var tail: Node<T>? = null
8     public override var size: Int = 0
9
10    public class Node<T>(var value: T) {
11        var next: Node<T>? = null
12    }
13
14    public fun enqueue(item: T) {
15
16        val node = Node(item)
17        val tail = this.tail
18
19        if (tail == null) {
20            this.head = node
21            this.tail = node
22        } else {
23            tail.next = node
24            this.tail = node
25        }
26        size++
27    }
28
29    public fun dequeue() {
30        if (head == null)
31            return
32        head = head?.next
33    }
34
35    public fun peek(): T {
36        if (size == 0) throw NoSuchElementException()
37        return head!!.value
38    }
39
40    public override fun isEmpty(): Boolean {
41        return size == 0
42    }
43
44    public override fun contains(element: T): Boolean {
45        for (obj in this) {
46            if (obj == element) return true
47        }
48        return false
49    }
50
51    public override fun containsAll(elements: Collection<T>): Boolean {
52        for (element in elements) {

```

```

53         if (!contains(element)) return false
54     }
55     return true
56 }
57
58 public override fun iterator(): Iterator<T> {
59
60     return object : Iterator<T> {
61         var node = head
62
63         override fun hasNext(): Boolean {
64             return node != null
65         }
66
67         override fun next(): T {
68             if (!hasNext()) throw NoSuchElementException()
69
70             val current = node!!
71             node = current.next
72             return current.value
73         }
74     }
75 }
76 }

```

1.3 Linked List

LISTING 1.3 – Stack.

```

1 package dsa
2
3
4 class Node<T>(value: T){
5     var value: T = value
6     var next: Node<T>? = null
7     var prev: Node<T>? = null
8 }
9
10 class LinkedList<T> {
11
12     private var head:Node<T>? = null
13
14     var isEmpty: Boolean = (head == null)
15
16     fun first(): Node<T>? = head
17
18     fun last(): Node<T>? {
19         var node = head
20         if (node != null){
21             while (node?.next != null) {
22                 node = node?.next
23             }
24             return node
25         } else {
26             return null
27         }
28     }
29
30     fun count(): Int {
31         var node = head
32         if (node != null){
33             var counter = 1
34             while (node?.next != null){
35                 node = node?.next

```

```

36         counter += 1
37     }
38     return counter
39 } else {
40     return 0
41 }
42 }
43
44 fun nodeAtIndex(index: Int) : Node<T>? {
45     if (index >= 0) {
46         var node = head
47         var i = index
48         while (node != null) {
49             if (i == 0) return node
50             i -= 1
51             node = node.next
52         }
53     }
54     return null
55 }
56
57 fun append(value: T) {
58     var newNode = Node(value)
59
60     var lastNode = this.last()
61     if (lastNode != null) {
62         newNode.prev = lastNode
63         lastNode.next = newNode
64     } else {
65         head = newNode
66     }
67 }
68
69 fun removeAll() {
70     head = null
71 }
72
73 fun removeNode(node: Node<T>): T {
74     val prev = node.prev
75     val next = node.next
76
77     if (prev != null) {
78         prev.next = next
79     } else {
80         head = next
81     }
82     next?.prev = prev
83
84     node.prev = null
85     node.next = null
86
87     return node.value
88 }
89
90 fun removeLast() : T? {
91     val last = this.last()
92     if (last != null) {
93         return removeNode(last)
94     } else {
95         return null
96     }
97 }
98

```



```
99     fun removeAtIndex(index: Int): T? {
100         val node = nodeAtIndex(index)
101         if (node != null) {
102             return removeNode(node)
103         } else {
104             return null
105         }
106     }
107
108     override fun toString(): String {
109         var s = "["
110         var node = head
111         while (node != null) {
112             s += "${node.value}"
113             node = node.next
114             if (node != null) { s += ", " }
115         }
116         return s + "]"
117     }
118 }
```


SEARCH

2.1 Linear Search

LISTING 2.1 – Linear Search.

```
1 public class SEA_Linear {
2
3     fun <T : Comparable<T>> linearSearch(list: List<T>, key: T): Int? {
4         for ((index, value) in list.withIndex()) {
5             if (value == key) return index
6         }
7         return null
8     }
9 }
```

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    }
23 }
```


SORT

3.1 Bubble Sort

LISTING 3.1 – Bubble Sort.

```
1 public fun bubblesort(inp: IntArray) {  
2     var swappedElements : Boolean;  
3     do {  
4         swappedElements = false;  
5         for (i in 0..inp.size - 2){  
6             if (inp[i] > inp[i + 1]){  
7                 val tmp = inp[i+1]  
8                 inp[i+1] = inp[i]  
9                 inp[i] = tmp  
10                swappedElements = true;  
11            }  
12        }  
13    } while (swappedElements);  
14 }
```

3.2 Insertion Sort

LISTING 3.2 – Insertion Sort.

```
1 fun insertionSort(inp: IntArray) {  
2  
3     for (i in 1..(inp.size)-1) {  
4         val value = inp[i]  
5         var hole = i  
6         while ( (hole > 0) &&  
7             (inp[hole-1] > value)) {  
8             inp[hole] = inp[hole - 1]  
9             hole = hole - 1  
10        }  
11        inp[hole] = value  
12    }  
13 }
```

3.3 Merge Sort

LISTING 3.3 – Merge Sort.

```

1 fun mergeSort(list: List<Int>): List<Int> {
2     if (list.size <= 1) {
3         return list
4     }
5
6     val middle = list.size / 2
7     var left = list.subList(0, middle);
8     var right = list.subList(middle, list.size);
9
10    return merge(mergeSort(left), mergeSort(right))
11 }
12
13 fun merge(left: List<Int>, right: List<Int>): List<Int> {
14     var indexLeft = 0
15     var indexRight = 0
16     var newList: MutableList<Int> = mutableListOf()
17
18     while (indexLeft < left.count() && indexRight < right.count()) {
19         if (left[indexLeft] <= right[indexRight]) {
20             newList.add(left[indexLeft])
21             indexLeft++
22         } else {
23             newList.add(right[indexRight])
24             indexRight++
25         }
26     }
27
28     while (indexLeft < left.size) {
29         newList.add(left[indexLeft])
30         indexLeft++
31     }
32
33     while (indexRight < right.size) {
34         newList.add(right[indexRight])
35         indexRight++
36     }
37
38     return newList;
39 }

```

3.4 Qsort

LISTING 3.4 – QSort.

```

1 fun <T: Comparable<T>> quicksort(items: List<T>): List<T> {
2     if (items.count() < 2) {
3         return items
4     }
5     val pivot = items[items.count() / 2]
6     val equal = items.filter { it == pivot }
7     val less = items.filter { it < pivot }
8     val greater = items.filter { it > pivot }
9     return quicksort(less) + equal + quicksort(greater)
10 }

```

3.5 Selection Sort

LISTING 3.5 – Selection Sort.

```

1 fun <T : Comparable<T>> selectionSort(items: MutableList<T>):
2     MutableList<T> {
3
4     if (items.isEmpty()) {
5         return items
6     }
7
8     for (idx in 0..items.count()) {
9         val array = items.subList(0, items.count() - idx)
10        val minItem = array.min()
11        val indexOfMinItem = array.indexOf(minItem)
12
13        if (minItem != null) {
14            items.removeAt(indexOfMinItem)
15            items.add(minItem)
16        }
17    }
18
19    for (x in items)
20        println("$x ")
21    println()
22    return items
23 }

```

3.6 Shell Sort

LISTING 3.6 – Shell Sort.

```

1 fun shellSort(inp: IntArray) {
2     var gap = 1
3     var value: Int
4     var index: Int
5
6     while (gap < inp.size / 3) {
7         gap = 3 * gap + 1
8     }
9
10    while (gap > 0) {
11
12        for (i in gap until inp.size step 1) {
13            value = inp[i]
14            index = i
15
16            while ( (index >= gap) &&
17                (value < inp[index - gap]) ) {
18
19                inp[index] = inp[index - gap]
20                index = index - gap
21                readLine()
22            }
23
24            inp[index] = value
25        }
26
27        gap = (gap - 1) / 3
28    }
29
30    for (x in inp)
31        print("$x ")
32    println()
33 }

```


BINARY TREE

4.1 Depth First Search

LISTING 4.1 – Depth First Search.

```
1 package dsa
2
3 import java.util.*
4
5 public class N(var value: Int, var left: N?, var right: N?)
6
7 class TRE_Traverse {
8
9     fun preorder(root: N?) {
10         if (root == null)
11             return
12         print("${root.value} ")
13         if (root.left != null)
14             preorder(root!!.left)
15         if (root.right != null)
16             preorder(root!!.right)
17     }
18
19     fun inorder(root: N?) {
20         if (root == null)
21             return
22         if (root.left != null)
23             inorder(root.left)
24         print("${root.value} ")
25         if (root.right != null)
26             inorder(root.right)
27     }
28
29     fun postorder(root: N?) {
30         if (root == null)
31             return
32         if (root.left != null)
33             postorder(root.left)
34         if (root.right != null)
35             postorder(root.right)
36         print("${root.value} ")
37     }
38 }
```

4.2 Breadth First Search

LISTING 4.2 – Breadth First Search.

```

1 fun bfs(root: N) {
2     val q: Queue<N> = ArrayDeque<N>()
3
4     if (root == null)
5         return;
6
7     q.add(root)
8     while (!q.isEmpty()) {
9         val current = q.remove()
10        print("${current.value} ")
11        if (current.left != null)
12            q.add(current.left)
13        if (current.right != null)
14            q.add(current.right)
15    }
16    println()
17 }

```

4.3 IS Binary Search Tree

LISTING 4.3 – Is BST.

```

1 fun isBST(root: N) : Boolean {
2     if (isSubTreeLesser(root.left!!, root.value) &&
3         isSubTreeGreater(root.right!!, root.value) &&
4         isBST(root.left!!) &&
5         isBST(root.right!!)) {
6         return(true)
7     }
8     else
9         return(false)
10 }
11
12 fun isSubTreeLesser(root: N, value: Int) : Boolean {
13     if (root == null)
14         return(true)
15
16     if ( (root.value < value) &&
17         isSubTreeLesser(root.left!!,value) &&
18         isSubTreeGreater(root.right!!,value)) {
19         return(true)
20     }
21     else
22         return(false)
23 }
24
25 fun isSubTreeGreater(root: N, value: Int) : Boolean {
26     if (root == null)
27         return(true)
28     if ( (root.value > value) &&
29         isSubTreeLesser(root.left!!,value) &&
30         isSubTreeGreater(root.right!!,value)) {
31         return(true)
32     }
33     else
34         return(false)
35 }

```

4.4 Insert, Delete Node

LISTING 4.4 – Insert, Delete Node.

```

1 fun insertNode(root: N?, key: Int): N {
2     var root = root
3
4     if (root == null) { // If the tree is empty, return a new node
5         return root!!
6     }
7
8     /* Otherwise, recur down the tree */
9     if (key < root.value)
10        root.left = insertNode(root.left!!, key)
11    else if (key > root.value)
12        root.right = insertNode(root.right!!, key)
13
14    return root!! // return the (unchanged) node pointer
15 }
16
17 // A recursive function to insert a new key in BST
18 fun deleteNode(root: N?, key: Int): N? {
19
20     // If the tree is empty, return
21     if (root == null)
22         return root
23
24     // Otherwise, recur down the tree
25     if (key < root.value)
26         root.left = deleteNode(root.left, key)
27
28     else if (key > root.value)
29         root.right = deleteNode(root.right, key)
30
31     else { // if key is same as root's key, then delete node
32         // node with only one child or no child
33         if (root.left == null)
34             return root.right
35
36         else if (root.right == null)
37             return root.left
38
39         // node with 2 children;
40         // Get inorder successor (smallest in right subtree)
41         root.value = minValue(root.right)
42
43         // Delete the inorder successor
44         root.right = deleteNode(root.right, root.value)
45     }
46
47     return root
48 }
49
50 fun minValue(root: N?): Int {
51
52     var root = root
53     var minv = root!!.value
54
55     while (root?.left!! != null) {
56         minv = root?.left!!.value
57         root = root?.left!!
58     }
59     return minv
60 }

```

4.5 Sorted List to BST

LISTING 4.5 – Sorted List to BST.

```
1 fun sortedArrayToBST(arr: IntArray, start: Int, end: Int): Node? {  
2  
3     if (start > end) {  
4         return null  
5     }  
6  
7     // Get the middle element and make it root  
8     val mid = (start + end) / 2  
9     val node = Node(arr[mid])  
10  
11     // Recursively construct left subtree; make it left child of root  
12     node.left = sortedArrayToBST(arr, start, mid - 1)  
13  
14     // Recursively construct right subtree; make it right child of root  
15     node.right = sortedArrayToBST(arr, mid + 1, end)  
16  
17     return node  
18 }
```

MINIMUM SPANNING TREE

5.1 Kruskal'S Algorithm

```

1 package kruskal
2
3 public class MSP_Kruskal {
4     var mlSeg = mutableListOf<Seg>()
5
6     init {
7
8         mlSeg.add(Seg(0,0,1,10))
9         mlSeg.add(Seg(1,1,2, 8))
10        mlSeg.add(Seg(2,2,3, 7))
11        mlSeg.add(Seg(3,3,4, 9))
12        mlSeg.add(Seg(4,4,5,10))
13        mlSeg.add(Seg(5,5,6, 2))
14        mlSeg.add(Seg(6,6,7, 1))
15        mlSeg.add(Seg(7,7,0, 8))
16    }
17
18    fun kruskal() {
19        var msp = mutableListOf<Seg>()
20
21        mlSeg.sortBy { it.ds }
22
23        var mspSize = 0
24        while (true) {
25
26            var iterator = mlSeg.listIterator() // To modify collection
27
28            while (iterator.hasNext()) {
29                var seg = iterator.next()
30
31                if (!(isCycle(msp, seg))) {
32                    iterator.remove()
33                    msp.add(seg)
34                    mspSize++
35                }
36            }
37
38            if (msp.size == mspSize)
39                break
40        }
41
42        printColl(msp)
43    }
44
45    fun isCycle(msp: List<Seg>, seg: Seg): Boolean {
46
47        val con1 = msp.filter { (it.n1 == seg.n1) || (it.n2 == seg.n1) }.
48            count()
49        val con2 = msp.filter { (it.n1 == seg.n2) || (it.n2 == seg.n2) }.
50            count()
51
52        if (( con1 == 0) && (con2 == 0) ) {

```

```
53         return false
54     }
55     else if ( ( (con1 != 0) && (con2 == 0) ) ||
56              ( (con1 == 0) && (con2 != 0) ) ) {
57         return false
58     }
59     else {
60         return true
61     }
62 }
63 }
64 }
65 }
66
67 data class Seg (val id: Int, val n1: Int, val n2: Int, val ds: Int)
68
69 fun printColl(coll: List<Any>) {
70     for (c in coll)
71         println(c)
72     println()
73 }
```

GRAPH TRAVERSAL

6.1 Dijkstra's Algorithm

The algorithm 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.

```

1 package djikstra
2
3 public class Graph {
4     var mlVisited = mutableListOf<Int>()
5     var mlUnvisited = mutableListOf<Int>()
6
7     var mlConn = mutableListOf<List<Int>>()
8
9     var mlCost = mutableListOf<Cost>()
10
11     init {
12         mlUnvisited.add(0)
13         mlUnvisited.add(1)
14         mlUnvisited.add(2)
15         mlUnvisited.add(3)
16         mlUnvisited.add(4)
17
18         mlConn.add(mutableListOf( 0, 6, -1, 1, -1))
19         mlConn.add(mutableListOf( 6, 0, 5, 2, 2))
20         mlConn.add(mutableListOf(-1, 5, 0, -1, 5))
21         mlConn.add(mutableListOf( 1, 2, -1, 0, 1))
22         mlConn.add(mutableListOf(-1, 2, 5, 1, 0))
23
24         mlCost.add(Cost(0, 0, 0))
25         mlCost.add(Cost(1, Int.MAX_VALUE, -1))
26         mlCost.add(Cost(2, Int.MAX_VALUE, -1))
27         mlCost.add(Cost(3, Int.MAX_VALUE, -1))
28         mlCost.add(Cost(4, Int.MAX_VALUE, -1))
29     }
30
31     fun traverse(startFrom: Int) {
32         var from = startFrom
33
34         while (mlUnvisited.isNotEmpty()) {
35             var listOfNodeDistPair = getListOfNodeDistPair(from)
36             var listOfNodeDistPairUnvisited =
37                 listOfNodeDistPair.filter { (n,d) -> !(n in mlVisited) }

```

```

38         if ( listOfNodeDistPairUnvisited.isNotEmpty() ) {
39             updateCost(from, listOfNodeDistPairUnvisited)
40             mlVisited.add(from)
41             mlUnvisited.remove(from)
42             from = listOfNodeDistPairUnvisited.get(0).first
43         }
44     }
45     else
46         break
47 }
48 for (c in mlCost)
49     println(c)
50 }
51
52 // Get Node and Distance, sorted by distance
53 fun getListOfNodeDistPair(from: Int): List<Pair<Int,Int>> {
54
55     var mlND = mutableListOf<Pair<Int,Int>>()
56     val row = mlConn.get(from)
57
58     for (i in 0..row.size-1) {
59         val d = row.get(i)
60         if ((d != 0) && (d != -1)) {
61             mlND.add(Pair(i, d))
62         }
63     }
64     mlND.sortBy { (x,y) -> y }
65
66     return mlND
67 }
68
69 // Update Cost
70 fun updateCost(from: Int, listNodeDistPair: List<Pair<Int,Int>>) {
71
72     val costUptoFrom = mlCost.get(from).dist
73
74     for (p in listNodeDistPair) {
75
76         val to = p.first
77         val ds = p.second
78
79         val currCost = mlCost.get(to).dist
80         val newCost = costUptoFrom + ds
81
82         if (newCost < currCost) {
83             mlCost.get(to).dist = newCost
84             mlCost.get(to).prev = from
85         }
86     }
87 }
88 }
89 }
90
91 data class Conn(val n1: Int, val n2: Int, val dist: Int)
92 data class Cost(val node: Int, var dist: Int, var prev: Int)
93

```


GREEDY

7.1 Knapsack

LISTING 7.1 – Knapsack.

```

1 package dsa
2
3 class Knapsack {
4     val wants = listOf(
5         Item("map", 9, 150),
6         Item("compass", 13, 35),
7         Item("water", 153, 200),
8         Item("sandwich", 50, 160),
9         Item("glucose", 15, 60),
10        Item("tin", 68, 45),
11        Item("banana", 27, 60),
12        Item("apple", 39, 40)
13    )
14
15    val MAX_WEIGHT = 400
16
17    init {
18        val (chosenItems, totalWeight, totalValue) =
19            m(wants.size - 1, MAX_WEIGHT)
20        println("Knapsack Item Chosen Weight Value")
21        for (item in chosenItems.sortedByDescending { it.value } )
22            println("${item.name.padEnd(24)} ${"%3d".format(item.weight)}
23                ${"%3d".format(item.value)}")
24        println("Total ${chosenItems.size} Items Chosen $totalWeight
25            $totalValue")
26    }
27
28    fun m(i: Int, w: Int): Triple<MutableList<Item>, Int, Int> {
29        val chosen = mutableListOf<Item>()
30
31        if (i < 0 || w == 0)
32            return Triple(chosen, 0, 0)
33        else if (wants[i].weight > w)
34            return m(i - 1, w)
35
36        val (l0, w0, v0) = m(i - 1, w)
37        val (l1, w1, v1) = m(i - 1, w - wants[i].weight)
38
39        v1 += wants[i].value
40        if (v1 > v0) {
41            l1.add(wants[i])
42            return Triple(l1, w1 + wants[i].weight, v1)
43        }
44
45        return Triple(l0, w0, v0)
46    }
47
48    data class Item(val name: String, val weight: Int, val value: Int)
49 }

```


