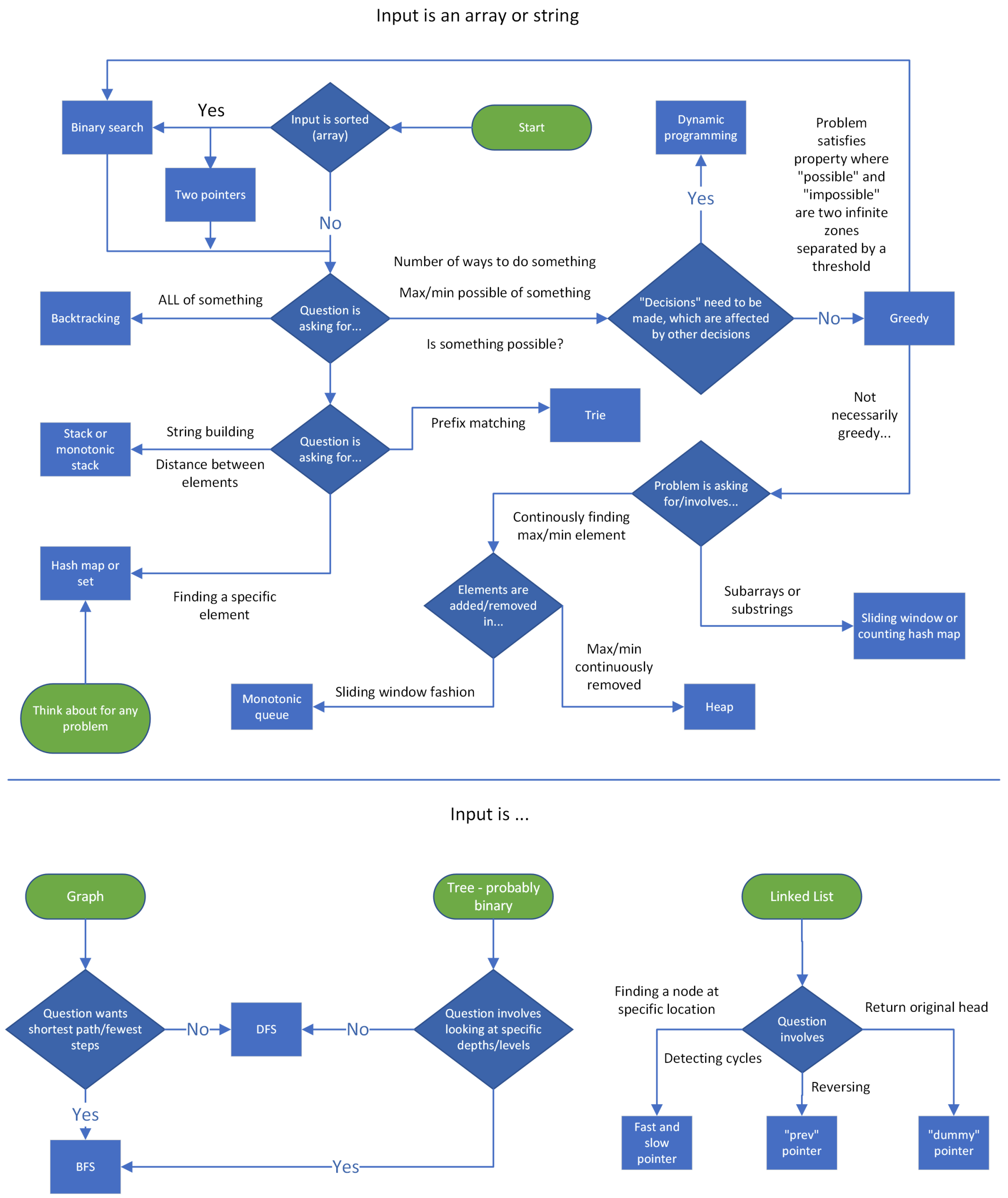
Data Structures & Algorithms Patterns

Code templetes for Data Structures and Algorithms Crash Course from LeetCode (with some edits from me).

Helper Flowchart

NOTE: This is a helper flowchart, it covers majority of problems. But, it is not possible to cover all possible problems.



```kotlin
for(i in 1..<10) { print("$i ") }

val stack = Stack<Int>()  ;  stack.add(5)  ;  stack.removeAt(0)
stack.push(8)  ;  = stack.pop()  ;  = stack.peek()

val queue = ArrayDeque<Int>()  ;  queue.add(5)

var mutableList = mutableListOf("Mahipal","Nikhil","Rahul")
mutableList[0] = "Praveen"
mutableList.add("Abhi")
for(item in mutableList) { println(item) }

var mutableSet = mutableSetOf<Int>(6,10)
mutableSet.add(2)
for(item in mutableSet){ println(item) }
mutableSet.contains(str);  mutableSet.remove("B");

var mutableMap = mutableMapOf<Int,String>(1 to "Neha",2 to "Puja")
mutableMap.put(1,"Rani")
mutableMap.put(4,"Abhi")

for(value in mutableMap.values){ println(value) }
for(key in immutableMap.keys){ println(immutableMap[key]) }

map.put(100,"Amit");     map.remove(102);
map.containsKey(5);     map.containsValue("World");
```

```kotlin
fun main() {
    var myList = LinkedList()
    myList.addAtHead(99)
    myList.addAtHead("kkkk")
    println(myList.get(1))
}

class LinkedList {
    var head: Node? = null
    var tail: Node? = null
    var length: Int = 0
    inner class Node(var value: Any?){
        var next: Node? = null
    }

    fun addAtHead(value: Any?){
        val h = this.head
        val newNode = Node(value)
        newNode.next = this.head
        head = newNode
        if (h == null) tail = newNode
        this.length++
    }

    fun addAtTail(value: Any?){
        var h = head
        val newNode = Node(value)
        newNode.next = null
        while (h!!.next !=null) h = h.next
        h.next = newNode
        tail = newNode
        this.length++
    }

    fun addAtIndex(index: Int, value: Any?){
        var h = head
        var newNode = Node(value)
        var counter = 0
        if (index < 0 || index > this.length) return
        if (index == 0) {
            addAtHead(value)
            return
        }
        if (index == this.length) {
            addAtTail(value)
            return
        }
        while (counter != index-1){
            h = h!!.next
            counter++
        }
        newNode.next = h!!.next
```

```
        h.next = newNode
        this.length++
    }

    fun deleteAtIndex(index: Int) {
        var curr = this.head
        var prev:Node? = null
        var counter = 0
        if (index < 0 || index >= this.length) return
        if (index == 0){
            head = curr!!.next
            this.length--
            return
        }
        while (counter != index){
            prev = curr
            curr = prev!!.next
            counter++
        }
        prev!!.next = curr!!.next
        if (index == length-1) tail = prev
        this.length--
    }

    fun get(index: Int): Any?{
        var h = head
        var counter = 0
        if (index < 0 || index >= this.length) return -1
        while (counter != index){
            h = h!!.next
            counter++
        }
        return h!!.value
    }

}
```

```
java.util.*

**Types:** Boolean char-Character Byte Short int-Integer Long Float

int[] ar = new int[5]; int ar[] = {3, 1, 9, 2};
int c[][]=new int[2][3]; int a[][]={{1,3,4},{3,4,5}};
**Array Class:** List<Integer> l1 = Arrays.asList(ar);
Arrays.sort(ar); Arrays.binarySearch(ar,9);

List<String> al = new ArrayList<String>();
int size = al.size(); al.add("Ravi");  al.remove(0);   //index

Set<String> hs = new HashSet<String>();
hs.add("A");  hs.contains(str);  hs.remove("B");
for (String val:hs) { println(val);}

Map<Integer,String> map = new HashMap<Integer,String>();
map.put(100,"Amit");     map.remove(102);
map.containsKey(5);     map.containsValue("World");
for(Map.Entry m:map.entrySet()){  println(m.getKey()+" "+m.getValue

Collections.max(al);  Collections.min(al);   Collections.sort(al)

length can be used for int[], double[], String[]

**String Class:** i length(), ch charAt(i ind), bo contains(chSeq s
bo equals(Obj another), sr replace(ch old, ch new), sr trim()
sr[] split(sr regex), i indexOf(i ch), toLowerCase()

public class ListNode {
int val; ListNode next;
ListNode(int x) { val = x; }

public ListNode reverseList(ListNode head) {
     if(head == null || head.next == null)   return head;
     -ListNode newHead=reverseList(head.next);
     -head.next.next=head;  head.next=null;
     return newHead;}
```

1) Array - Two pointers: one input, opposite ends

```java
public int fn(int[] arr) {
    int left = 0;
    int right = arr.length - 1;
    int ans = 0;

    while (left < right) {
        // do some logic here with left and right
        if (CONDITION) {
            left++;
        } else {
            right--;
        }
    }

    return ans;
}
```

2) Array - Two pointers: two inputs, exhaust both

```java
public int fn(int[] arr1, int[] arr2) {
    int i = 0, j = 0, ans = 0;

    while (i < arr1.length && j < arr2.length) {
        // do some logic here
        if (CONDITION) {
            i++;
        } else {
            j++;
        }
    }

    while (i < arr1.length) {
        // do logic
        i++;
    }

    while (j < arr2.length) {
        // do logic
        j++;
    }

    return ans;
}
```

3) Array - Sliding window

```java
public int fn(int[] arr) {
    int left = 0, ans = 0, curr = 0;

    for (int right = 0; right < arr.length; right++) {
        // do logic here to add arr[right] to curr

        while (WINDOW_CONDITION_BROKEN) {
            // remove arr[left] from curr
            left++;
        }

        // update ans
    }

    return ans;
}
```

4) Array - Build a prefix sum

```java
public int[] fn(int[] arr) {
    int[] prefix = new int[arr.length];
    prefix[0] = arr[0];

    for (int i = 1; i < arr.length; i++) {
        prefix[i] = prefix[i - 1] + arr[i];
    }

    return prefix;
}
```

5) Efficient string building

```java
public String fn(char[] arr) {
    StringBuilder sb = new StringBuilder();
    for (char c: arr) {
        sb.append(c);
    }

    return sb.toString();
}
```

6) Find number of subarrays that fit an exact criteria

```java
public int fn(int[] arr, int k) {
    Map<Integer, Integer> counts = new HashMap<>();
    counts.put(0, 1);
    int ans = 0, curr = 0;

    for (int num: arr) {
        // do logic to change curr
        ans += counts.getOrDefault(curr - k, 0);
        counts.put(curr, counts.getOrDefault(curr, 0) + 1);
    }

    return ans;
}
```

7) Binary search

```java
public int fn(int[] arr, int target) {
    int left = 0;
    int right = arr.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            // do something
            return mid;
        }
        if (arr[mid] > target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    // left is the insertion point
    return left;
}
```

8) Monotonic increasing stack

The same logic can be applied to maintain a monotonic queue.

```java
public int fn(int[] arr) {
    Stack<Integer> stack = new Stack<>();
    int ans = 0;

    for (int num: arr) {
        // for monotonic decreasing, just flip the > to <
        while (!stack.empty() && stack.peek() > num) {
            // do logic
            stack.pop();
        }

        stack.push(num);
    }

    return ans;
}
```

9) Find top k elements with heap

```java
public int[] fn(int[] arr, int k) {
    PriorityQueue<Integer> heap = new PriorityQueue<>(CRITERIA);
    for (int num: arr) {
        heap.add(num);
        if (heap.size() > k) {
            heap.remove();
        }
    }

    int[] ans = new int[k];
    for (int i = 0; i < k; i++) {
        ans[i] = heap.remove();
    }

    return ans;
}
```

10) Linked list: fast and slow pointer

```java
public int fn(ListNode head) {
    ListNode slow = head;
    ListNode fast = head;
    int ans = 0;

    while (fast != null && fast.next != null) {
        // do logic
        slow = slow.next;
        fast = fast.next.next;
    }

    return ans;
}
```

11) Reversing a linked list

```java
public ListNode fn(ListNode head) {
    ListNode curr = head;
    ListNode prev = null;
    while (curr != null) {
        ListNode nextNode = curr.next;
        curr.next = prev;
        prev = curr;
        curr = nextNode;
    }

    return prev;
}
```

12) Binary tree: DFS (recursive)

```java
public int dfs(TreeNode root) {
    if (root == null) {
        return 0;
    }

    int ans = 0;
    // do logic
    dfs(root.left);
    dfs(root.right);
    return ans;
}
```

13) Binary tree: DFS (iterative)

```java
public int dfs(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    stack.push(root);
    int ans = 0;

    while (!stack.empty()) {
        TreeNode node = stack.pop();
        // do logic
        if (node.left != null) {
            stack.push(node.left);
        }
        if (node.right != null) {
            stack.push(node.right);
        }
    }

    return ans;
}
```

14) Binary tree: BFS (iterative)

```java
public int fn(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    int ans = 0;

    while (!queue.isEmpty()) {
        int currentLength = queue.size();
        // do logic for current level

        for (int i = 0; i < currentLength; i++) {
            TreeNode node = queue.remove();
            // do logic
            if (node.left != null) {
                queue.add(node.left);
            }
            if (node.right != null) {
                queue.add(node.right);
            }
        }
    }

    return ans;
}
```

15) Graph: DFS (recursive)

For the graph templates, assume the nodes are numbered from 0 to n - 1 and the graph is given as an adjacency list.

Depending on the problem, you may need to convert the input into an equivalent adjacency list before using the templates.

```
Set<Integer> seen = new HashSet<>();

public int fn(int[][] graph) {
    seen.add(START_NODE);
    return dfs(START_NODE, graph);
}

public int dfs(int node, int[][] graph) {
    int ans = 0;
    // do some logic
    for (int neighbor: graph[node]) {
        if (!seen.contains(neighbor)) {
            seen.add(neighbor);
            ans += dfs(neighbor, graph);
        }
    }

    return ans;
}
```

16) Graph: DFS (iterative)

```java
public int fn(int[][] graph) {
    Stack<Integer> stack = new Stack<>();
    Set<Integer> seen = new HashSet<>();
    stack.push(START_NODE);
    seen.add(START_NODE);
    int ans = 0;

    while (!stack.empty()) {
        int node = stack.pop();
        // do some logic
        for (int neighbor: graph[node]) {
            if (!seen.contains(neighbor)) {
                seen.add(neighbor);
                stack.push(neighbor);
            }
        }
    }

    return ans;
}
```

17) Graph: BFS (iterative)

```java
public int fn(int[][] graph) {
    Queue<Integer> queue = new LinkedList<>();
    Set<Integer> seen = new HashSet<>();
    queue.add(START_NODE);
    seen.add(START_NODE);
    int ans = 0;

    while (!queue.isEmpty()) {
        int node = queue.remove();
        // do some logic
        for (int neighbor: graph[node]) {
            if (!seen.contains(neighbor)) {
                seen.add(neighbor);
                queue.add(neighbor);
            }
        }
    }

    return ans;
}
```

18) Binary search: duplicate elements, left-most insertion point

```java
public int fn(int[] arr, int target) {
    int left = 0;
    int right = arr.length;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] >= target) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return left;
}
```

19) Binary search: duplicate elements, right-most insertion point

```java
public int fn(int[] arr, int target) {
    int left = 0;
    int right = arr.length;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] > target) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return left;
}
```

20) Binary search: for greedy problems - looking for minimum

```java
public int fn(int[] arr) {
    int left = MINIMUM_POSSIBLE_ANSWER;
    int right = MAXIMUM_POSSIBLE_ANSWER;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (check(mid)) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return left;
}

public boolean check(int x) {
    // this function is implemented depending on the problem
    return BOOLEAN;
}
```

21) Binary search: for greedy problems - looking for maximum

```java
public int fn(int[] arr) {
    int left = MINIMUM_POSSIBLE_ANSWER;
    int right = MAXIMUM_POSSIBLE_ANSWER;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (check(mid)) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return right;
}

public boolean check(int x) {
    // this function is implemented depending on the problem
    return BOOLEAN;
}
```

22) Backtracking problems

```java
public int backtrack(STATE curr, OTHER_ARGUMENTS...) {
    if (BASE_CASE) {
        // modify the answer
        return 0;
    }

    int ans = 0;
    for (ITERATE_OVER_INPUT) {
        // modify the current state
        ans += backtrack(curr, OTHER_ARGUMENTS...)
        // undo the modification of the current state
    }
}
```

23) Dynamic programming: top-down memoization

```java
Map<STATE, Integer> memo = new HashMap<>();

public int fn(int[] arr) {
    return dp(STATE_FOR_WHOLE_INPUT, arr);
}

public int dp(STATE, int[] arr) {
    if (BASE_CASE) {
        return 0;
    }

    if (memo.contains(STATE)) {
        return memo.get(STATE);
    }

    int ans = RECURRENCE_RELATION(STATE);
    memo.put(STATE, ans);
    return ans;
}
```

24) Build a trie

```java
// note: using a class is only necessary if you want to store data (
// otherwise, you can implement a trie using only hash maps.
class TrieNode {
    // you can store data at nodes if you wish
    int data;
    Map<Character, TrieNode> children;
    TrieNode() {
        this.children = new HashMap<>();
    }
}

public TrieNode buildTrie(String[] words) {
    TrieNode root = new TrieNode();
    for (String word: words) {
        TrieNode curr = root;
        for (char c: word.toCharArray()) {
            if (!curr.children.containsKey(c)) {
                curr.children.put(c, new TrieNode());
            }

            curr = curr.children.get(c);
        }

        // at this point, you have a full word at curr
        // you can perform more logic here to give curr an attribute
    }

    return root;
}
```

25) Dijkstra's algorithm

```java
int[] distances = new int[n];
Arrays.fill(distances, Integer.MAX_VALUE);
distances[source] = 0;

Queue<Pair<Integer, Integer>> heap = new PriorityQueue<Pair<Integer,
heap.add(new Pair(0, source));

while (!heap.isEmpty()) {
    Pair<Integer, Integer> curr = heap.remove();
    int currDist = curr.getKey();
    int node = curr.getValue();

    if (currDist > distances[node]) {
        continue;
    }

    for (Pair<Integer, Integer> edge: graph.get(node)) {
        int nei = edge.getKey();
        int weight = edge.getValue();
        int dist = currDist + weight;

        if (dist < distances[nei]) {
            distances[nei] = dist;
            heap.add(new Pair(dist, nei));
        }
    }
}
```