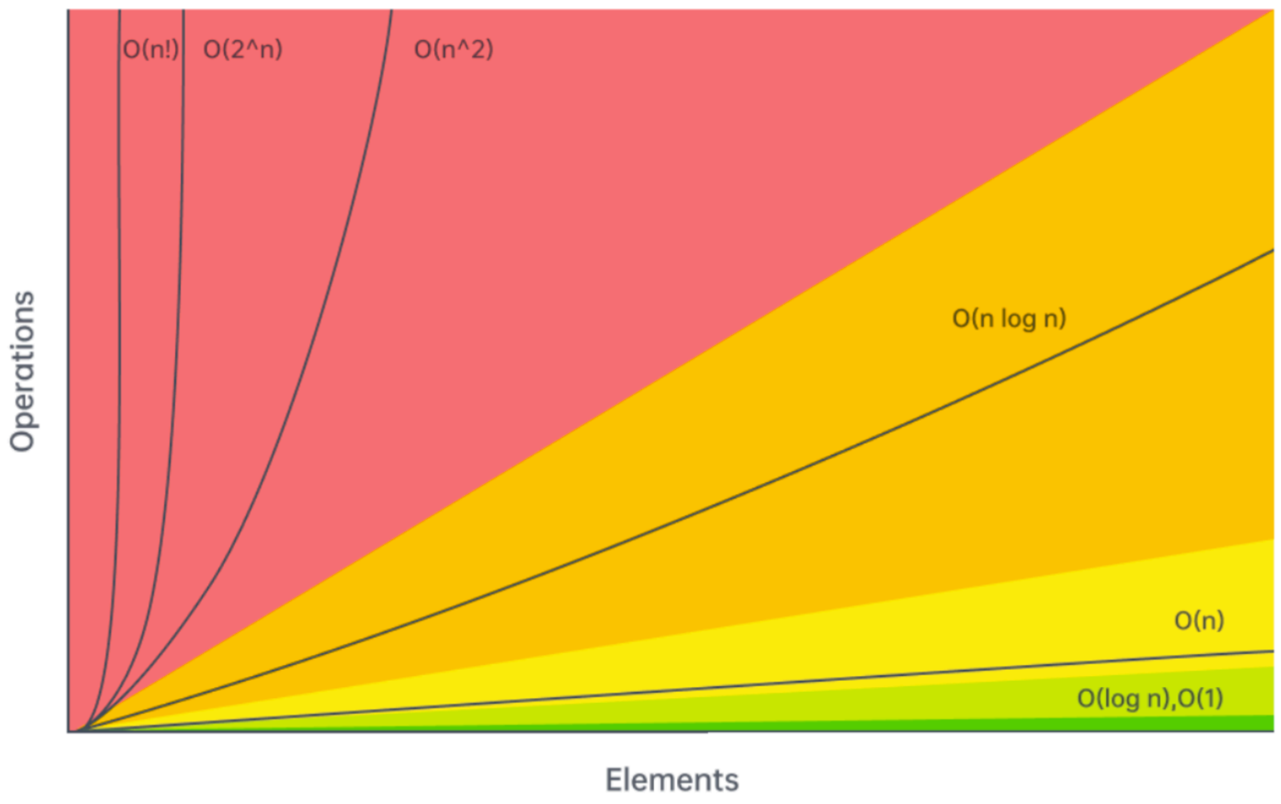
Leetcode Notes

Time Complexity



Arrays (dynamic array/list)

Given n = arr.length,

Add or remove element at the end: O(1) amortized
Add or remove element from arbitrary index: O(n)
Access or modify element at arbitrary index: O(1)
Check if element exists: O(n)
Two pointers: O(n·k), where k is the work done at each iteration, includes sliding window
Building a prefix sum: O(n)
Finding the sum of a subarray given a prefix sum: O(1)

Strings (immutable)

Given n = s.length,

Add or remove character: O(n)
Access element at arbitrary index: O(1)
Concatenation between two strings: O(n+m), where m is the length of the other string
Create substring: O(m), where m is the length of the substring
Two pointers: O(n·k), where k is the work done at each iteration, includes sliding window
Building a string from joining an array, stringbuilder, etc.: O(n)

Linked Lists

Given n as the number of nodes in the linked list,

Add or remove element given pointer before add/removal location: O(1)
Add or remove element given pointer at add/removal location:
O(1) if doubly linked
Add or remove element at arbitrary position without pointer: O(n)
Access element at arbitrary position without pointer: O(n)
Check if element exists: O(n)
Reverse between position i and j: O(j–i)
Detect a cycle: O(n) using fast-slow pointers or hash map

Hash table/dictionary

Given n = dic.length,

Add or remove key-value pair: O(1)
Check if key exists: O(1)
Check if value exists: O(n)
Access or modify value associated with key: O(1)
Iterate over all keys, values, or both: O(n)

Note: the O(1) operations are constant relative to n. In reality, the hashing algorithm might be expensive. For example, if your keys are strings, then it will cost O(m) where m is the length of the string. The operations only take constant time relative to the size of the hash map.

Set

Given n = set.length,

Add or remove element: O(1)
Check if element exists: O(1)
The above note applies here as well.

Stack

Stack operations are dependent on their implementation. A stack is only required to support pop and push. If implemented with a dynamic array:

Given n = stack.length,

Push element: O(1)
Pop element: O(1)
Peek (see element at top of stack): O(1)
Access or modify element at arbitrary index: O(1)
Check if element exists: O(n)

Queue

Queue operations are dependent on their implementation. A queue is only required to support dequeue and enqueue. If implemented with a doubly linked list:

Given n = queue.length,

Enqueue element: O(1)
Dequeue element: O(1)
Peek (see element at front of queue): O(1)
Access or modify element at arbitrary index: O(n)
Check if element exists: O(n)

Note: most programming languages implement queues in a more sophisticated manner than a simple doubly linked list. Depending on implementation, accessing elements by index may be faster than O(n), or O(n) but with a significant constant divisor.

Binary tree problems (DFS/BFS)

Given n as the number of nodes in the tree,

Most algorithms will run in O(n·k) time, where k is the work done at each node, usually O(1). This is just a general rule and not always the case. We are assuming here that BFS is implemented with an efficient queue.

Binary search tree

Given n as the number of nodes in the tree,

Add or remove element: O(n) worst case, O(logn) average case
Check if element exists: O(n) worst case, O(logn) average case

The average case is when the tree is well balanced - each depth is close to full. The worst case is when the tree is just a straight line.

Heap/Priority Queue

Given n = heap.length and talking about min heaps,

Add an element: O(logn)
Delete the minimum element: O(logn)
Find the minimum element: O(1)
Check if element exists: O(n)

Binary search

Binary search runs in O(logn) in the worst case, where n is the size of your initial search space.

Miscellaneous

Sorting: O(n·logn), where n is the size of the data being sorted

DFS and BFS on a graph: O(n·k+e), where n is the number of nodes, e is the number of edges, if each node is handled in O(1) other than iterating over edges
DFS and BFS space complexity: typically O(n), but if it's in a graph, might be O(n+e) to store the graph

Dynamic programming time complexity: O(n·k), where n is the number of states and k is the work done at each state
Dynamic programming space complexity: O(n), where n is the number of states

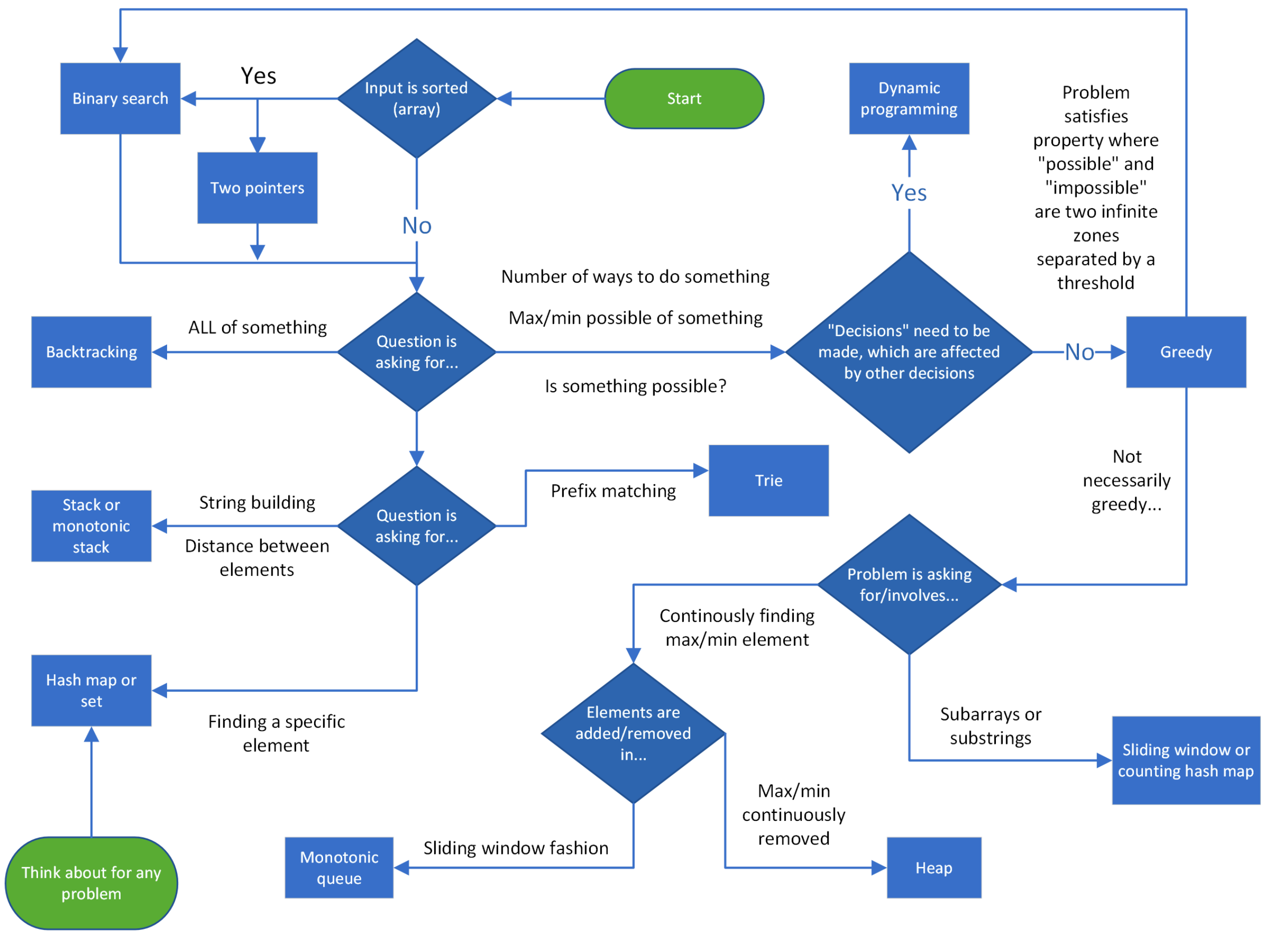
Sorting Algorithms

Sorting Algorithm	Time Complexity			Space Complexity	Stable
	Best	Average	Worst	Worst	
Quicksort	O(n log n)	O(n log n)	O(n^2)	O(log n)	No
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Heapsort	O(n log n)	O(n log n)	O(n log n)	O(1)	No
Timsort	O(n)	O(n log n)	O(n log n)	O(n)	Yes
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	No *
Shellsort	O(n log n)	-- (depends on gap sequence)	O(n^2)	O(1)	No
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	O(nk)	Yes
Radix Sort	O(nk)	O(nk)	O(nk)	O(nk)	Yes
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)	Yes

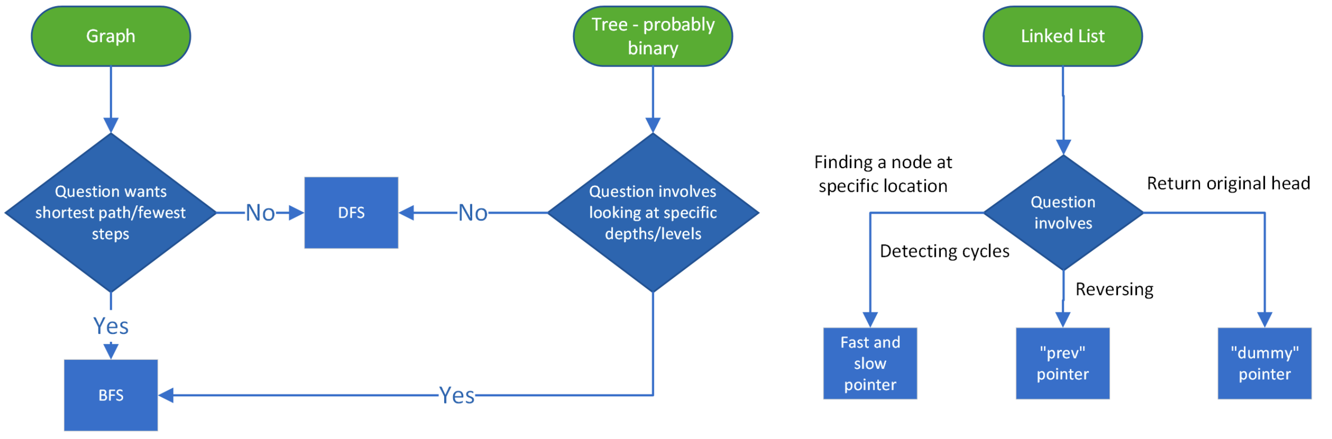
*Selection sort can be implemented as a stable sort if, rather than swapping the minimum value with its current value, the minimum value is inserted into the first position and the intervening values shifted up. However, this modification either requires a data structure that supports efficient insertions or deletions, such as a linked list, or it leads to O(n^2) writes.

DSA Flow-Chart

Input is an array or string


Start
Input is sorted (array)
Yes
Binary search
Two pointers
No
Question is asking for...
ALL of something
Backtracking
Number of ways to do something
Max/min possible of something
Is something possible?
"Decisions" need to be made, which are affected by other decisions
Yes
Dynamic programming
No
Greedy
Problem satisfies property where "possible" and "impossible" are two infinite zones separated by a threshold
Not necessarily greedy...
Question is asking for...
String building
Distance between elements
Stack or monotonic stack
Prefix matching
Trie
Finding a specific element
Hash map or set
Think about for any problem
Problem is asking for/involves...
Continously finding max/min element
Subarrays or substrings
Sliding window or counting hash map
Elements are added/removed in...
Sliding window fashion
Monotonic queue
Max/min continuously removed
Heap

Input is ...



Code Templates

Two pointers (1 input)

Problem Types: Finding pairs/elements with specific properties in a sorted array (e.g., sum equals a target), removing duplicates, reversing arrays, or checking for palindromes.

```
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;
}
```

Two pointers (2 inputs)

Problem Types: Merging two sorted arrays, finding intersections or unions of two arrays, comparing elements of two arrays to fulfill certain conditions.

```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;
}
```

Sliding window

Problem Types: Finding the longest/shortest substring or subarray meeting a certain condition, such as maximum sum, number of distinct characters, or other conditions that can be defined in a variable window size.

```
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;
}
```

Prefix sum

Problem Types: Calculating the sum of elements in a subarray, finding subarrays with a given sum, implementing range queries like sum or average in an array.

```
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;
}
```

Efficient string building

Problem Types: Constructing strings from characters or substrings, manipulating strings, and concatenating a large number of strings where direct concatenation would be inefficient.

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

    return sb.toString();
}
```

Linked list with pointers

Problem Types: Detecting cycles in a linked list, finding the middle of a linked list, and solving problems that involve slow and fast pointer techniques.

```
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;
}
```

Reversing a linked list

Problem Types: Reversing the nodes of a linked list, palindrome linked list check, or rearranging linked list nodes.

```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;
}
```

Find no of subarrays that fits certain criteria

Problem Types: Problems involving counting subarrays with a sum equal to or greater/less than a target, using cumulative sum and hashmap techniques for efficient computation.

```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;
}
```

Monotonic increasing stack

Problem Types: Finding the next greater/smaller elements, largest rectangle in a histogram, stock span problems, and other problems requiring elements processing in a specific order.

```
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;
}
```

Binary tree DFS (recursive)

Problem Types: Depth-first traversals, path sums, lowest common ancestor, and other tree-based recursive traversals and calculations.

```
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;
}
```

Binary tree DFS (interative)

Problem Types: Iterative depth-first search in trees, solving tree problems without recursion using a stack, such as inorder, preorder, postorder traversals.

```
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;
}
```

Binary tree BFS

Problem Types: Level order traversal, finding the minimum depth, maximum width of a tree, and other problems requiring processing nodes level by level.

```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;
}
```

Graph DFS (recursive)

Problem Types: Problems involving connected components, cycle detection, topological sort, and other depth-first searches in a graph.

```java
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;
}
```

Graph DFS (iterative)

Problem Types: Iterative implementations of graph problems typically solved recursively, like finding connected components or cycles in a graph.

```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;
}
```

Graph BFS

Problem Types: Shortest path in unweighted graphs, level order traversal in graphs, finding connected components, and other breadth-first search problems.

```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;
}
```

Find k elements with heap

Problem Types: Finding the k largest/smallest elements, k-closest points, and other problems requiring a dynamically maintained order of elements.

```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;
}
```

Binary search

Problem Types: Searching in sorted arrays, finding the first or last position of an element, and problems where the solution space can be halved at each step.

```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;
}
```

Binary search (duplicate elements, left most insertion point)

Problem Types: Finding the first occurrence of an element in a sorted array with duplicates, or binary search applied to a range of values.

```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;
}
```

Binary search (duplicate elements, right most insertion point)

Problem Types: Finding the last occurrence of an element in a sorted array with duplicates.

```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;
}
```

Binary search greedy (searching for minimum)

Problem Types: Optimization problems where you need to minimize a certain condition, often used in problems like finding the smallest/largest value under certain constraints.

```
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;
}
```

Binary search greedy (searching for maximum)

Problem Types: Similar to the above but for maximizing conditions.

```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;
}
```

Backtracking

Problem Types: Combinatorial problems like permutations, combinations, sudoku solver, N-queens, and other problems requiring exploring all possible solutions.

```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
    }
}
```

Dynamic programming top down memoization

Problem Types: Optimization problems, counting problems, and others where overlapping subproblems are solved recursively with memoization for efficiency.

```
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;
}
```

Dijkstra's algorithm

Problem Types: Finding the shortest path in weighted graphs, problems involving graph traversal with the least cost/weight.

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

Queue<Pair<Integer, Integer>> heap = new PriorityQueue<Pair<Integer,Integer>>(Comparator.
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));
        }
    }
}
```

Hash maps

Lost? Don't know what to do? Use a hash map!!

```java
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        // Creating a HashMap
        Map<String, Integer> map = new HashMap<>();

        // Adding key-value pairs to the HashMap
        map.put("Apple", 40);
        map.put("Banana", 10);
        map.put("Orange", 20);

        // Accessing a value
        System.out.println("Price of Apple: " + map.get("Apple"));

        // Iterating over key-value pairs
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }

        // Checking if a key exists
        if (map.containsKey("Banana")) {
            System.out.println("Banana is available");
        }

        // Removing a key-value pair
        map.remove("Orange");
    }
}
```