i have 1hr before my java interview. i want to learn most commonly dsa concepts. provide me quick guide and psuedocode to understand the concept.

Here's a rapid guide to the most commonly asked Data Structures & Algorithms (DSA) concepts in Java interviews, with explanations and pseudocode for each. Focus on these for your last-hour revision!

1. Arrays

Concept:
Collection of elements of the same type, indexed.

Key operations:

- Access: O(1)
- Insert/Delete: O(n) (except at end)

Pseudocode (Reverse an Array):

```
function reverseArray(arr):
    n = length(arr)
    for i from 0 to n/2 - 1:
        swap arr[i] with arr[n - i - 1]
```

2. Linked List

Concept:
A list of nodes, where each node contains data and reference to the next node.

Types:
Singly, Doubly, Circular.

Pseudocode (Traverse a Linked List):

```
function traverse(head):
    current = head
    while current is not null:
        print current.data
        current = current.next
```

3. Stack

Concept:
Last-In-First-Out (LIFO) structure. Operations: push, pop, peek.

Pseudocode:

```
Stack:
    list = []
    push(x):
        list.append(x)
    pop():
        if not empty:
            return list.pop()
    peek():
        return list[-1]
```

4. Queue

Concept:
First-In-First-Out (FIFO) structure. Operations: enqueue, dequeue.

Pseudocode:

```
Queue:
    list = []
    enqueue(x):
        list.append(x)
    dequeue():
        if not empty:
            return list.pop(0)
```

5. Binary Tree (esp. Binary Search Tree - BST)

Concept:
Each node has at most two children.
For BST: left child < parent < right child.

Pseudocode (Inorder Traversal):

```
function inorder(node):
    if node is not null:
        inorder(node.left)
        print node.data
        inorder(node.right)
```

6. HashMap (Hash Table)

Concept:
Maps keys to values for efficient data retrieval.

Operations:
Put, Get, Remove - all average O(1).

Pseudocode:

```
HashMap:
    array of buckets
    put(key, value):
        index = hash(key) % size
        add (key, value) to bucket[index]
    get(key):
        index = hash(key) % size
        search for key in bucket[index]
```

7. Sorting (Quick Sort Example)

Concept:
Arranges data in order.

Pseudocode:

```
function quickSort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)
```

8. Searching (Binary Search)

Concept:
Fast searching in sorted arrays: O(log n).

Pseudocode:

```
function binarySearch(arr, target):
    left = 0
    right = length(arr) - 1
    while left <= right:
        mid = (left + right) / 2
        if arr[mid] == target:
            return mid
        else if arr[mid] < target:
            left = mid + 1
        else:
```

```
            right = mid - 1
    return -1
```

9. Recursion

Concept:
Function calls itself to solve subproblems.

Pseudocode (Factorial):

```
function factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)
```

10. Big-O Notation

Concept:
Describes time/space complexity.

- O(1): Constant
- O(log n): Logarithmic
- O(n): Linear
- O(n log n): Linearithmic
- O(n^2): Quadratic

Tip for Interview:

- Know how to write and debug code on paper/whiteboard.
- Explain your solution, space/time complexity.

Good luck! If you need pseudocode for specific problems (e.g., cycle detection in a linked list, tree traversal), let me know!

also give me searching and sorting algos like this

Here's a concise guide to the most commonly asked **searching and sorting algorithms**, with pseudocode to help you grasp the essentials quickly.

Searching Algorithms

1. Linear Search

- **Use:** Find an element in an unsorted list.
- **Complexity:** O(n)

Pseudocode:

```
function linearSearch(arr, target):
    for i from 0 to length(arr) - 1:
        if arr[i] == target:
            return i
    return -1
```

2. Binary Search

- **Use:** Find an element in a **sorted** array.
- **Complexity:** O(log n)

Pseudocode:

```
function binarySearch(arr, target):
    left = 0
    right = length(arr) - 1
    while left <= right:
        mid = (left + right) / 2
        if arr[mid] == target:
            return mid
        else if arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Sorting Algorithms

1. Bubble Sort

- **Concept:** Repeatedly swap adjacent elements if they are in wrong order.
- **Complexity:** O(n^2)

Pseudocode:

```
function bubbleSort(arr):
    n = length(arr)
    for i from 0 to n-1:
        for j from 0 to n-i-2:
```

```
            if arr[j] > arr[j+1]:
                swap arr[j] and arr[j+1]
```

2. Selection Sort

- **Concept:** Repeatedly select the minimum element and place it at the beginning.
- **Complexity:** O(n^2)

Pseudocode:

```
function selectionSort(arr):
    n = length(arr)
    for i from 0 to n-1:
        minIndex = i
        for j from i+1 to n-1:
            if arr[j] < arr[minIndex]:
                minIndex = j
        swap arr[minIndex] and arr[i]
```

3. Insertion Sort

- **Concept:** Builds the sorted array one item at a time.
- **Complexity:** O(n^2)

Pseudocode:

```
function insertionSort(arr):
    n = length(arr)
    for i from 1 to n-1:
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j = j - 1
        arr[j+1] = key
```

4. Merge Sort

- **Concept:** Divide array into halves, sort them, then merge.
- **Complexity:** O(n log n)

Pseudocode:

```
function mergeSort(arr):
    if length(arr) <= 1:
        return arr
    mid = length(arr) / 2
```

```
    left = mergeSort(arr[0:mid])
    right = mergeSort(arr[mid:])
    return merge(left, right)

function merge(left, right):
    result = []
    while left and right:
        if left[0] < right:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    result.extend(left or right)
    return result
```

5. Quick Sort

- **Concept:** Partition the array around a pivot, then sort both halves.
- **Complexity:** O(n log n) average

Pseudocode:

```
function quickSort(arr):
    if length(arr) <= 1:
        return arr
    pivot = arr[length(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quickSort(left) + middle + quickSort(right)
```

If you need any particular algorithm explained with code or examples, just let me know!