DSA Algorithms - 5 to 6 Line Summary

Binary Search

```
1. Set low = 0, high = n-1
2. While low <= high:
3.     mid = (low + high) / 2
4.     If arr[mid] == key: return mid
5.     If arr[mid] < key: low = mid + 1
6.     Else: high = mid - 1
```

Quick Sort

1. Pick a pivot element.
2. Partition array around pivot.
3. Recursively sort left half.
4. Recursively sort right half.
5. Base case: 0 or 1 element.

Merge Sort

1. Divide array into 2 halves.
2. Recursively sort both halves.
3. Merge sorted halves.
4. Repeat until single element.
5. Return merged sorted array.

N-Queens

1. Place queen in each column.
2. Check for conflicts.
3. Recurse for next row.
4. Backtrack if needed.
5. Save board if row == N.

Activity Selection

1. Sort activities by end time.
2. Select first activity.
3. For each next activity:

```
4.    If start >= last end, select it.
5. Repeat for all.
```

0/1 Knapsack

```
1. Create dp[n+1][W+1] table.
2. Loop through items & weights.
3. If wt[i] <= w, take max(include, exclude).
4. Fill dp table accordingly.
5. Return dp[n][W].
```

BFS (Graph)

```
1. Initialize queue, mark visited.
2. While queue not empty:
3.    Dequeue node, process it.
4.    Enqueue unvisited neighbors.
5. Mark them visited.
```

DFS (Graph)

```
1. Start from source node.
2. Mark node visited.
3. Recurse for all neighbors.
4. Backtrack after finishing.
5. Continue until all visited.
```

Inorder Traversal

```
1. Traverse left subtree.
2. Visit current node.
3. Traverse right subtree.
```

Trie Insert

```
1. Start at root node.
2. For each character:
3.    If missing, create node.
4.    Move to next node.
5. Mark end of word.
```

Heapify

1. Compare parent with children.
2. Swap with largest/smallest.
3. Repeat down the tree.
4. Used to build heaps.

Count Set Bits

```
1. Initialize count = 0.
2. While n > 0:
3.     If n & 1, count++
4.     Right shift n.
5. Return count.
```

Max Sum Subarray (Size K)

1. Compute sum of first K.
2. Slide window: add next, drop prev.
3. Track max sum.
4. Repeat till end.

Balanced Parentheses

```
1. Initialize empty stack.
2. For each char:
3.     If opening, push.
4.     If closing, check top.
5. Stack empty means valid.
```

Reverse Linked List

```
1. prev = NULL, curr = head.
2. While curr != NULL:
3.     next = curr->next
4.     curr->next = prev
5.     prev = curr, curr = next
```