Yes, here are concise TypeScript templates for each of the patterns, designed to be easy to remember with clear comments.

1. Dynamic Programming

Fibonacci Numbers

TypeScript

```typescript
// Memoization (Top-Down)
const fibMemo = (n: number, memo: { [key: number]: number } = {}): number => {
  if (n in memo) return memo[n];
  if (n <= 1) return n;
  memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
  return memo[n];
};

// Tabulation (Bottom-Up)
const fibTab = (n: number): number => {
  if (n <= 1) return n;
  const dp: number[] = new Array(n + 1);
  dp[0] = 0;
  dp[1] = 1;
  for (let i = 2; i <= n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
  }
  return dp[n];
};
```

0/1 Knapsack

TypeScript

```typescript
const knapsack = (weights: number[], values: number[], capacity: number): number => {
  const n = weights.length;
  const dp: number[][] = Array(n + 1).fill(0).map(() => Array(capacity + 1).fill(0));

  for (let i = 1; i <= n; i++) {
    const weight = weights[i - 1];
    const value = values[i - 1];
    for (let w = 1; w <= capacity; w++) {
      if (weight <= w) {
        // Either exclude the item or include it
        dp[i][w] = Math.max(dp[i - 1][w], value + dp[i - 1][w - weight]);
      } else {
        // Must exclude the item
        dp[i][w] = dp[i - 1][w];
      }
    }
  }
  return dp[n][capacity];
};
```

Longest Common Subsequence

TypeScript

```typescript
const lcs = (text1: string, text2: string): number => {
  const m = text1.length;
  const n = text2.length;
  const dp: number[][] = Array(m + 1).fill(0).map(() => Array(n + 1).fill(0));
```

```
  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (text1[i - 1] === text2[j - 1]) {
        dp[i][j] = 1 + dp[i - 1][j - 1];
      } else {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
    }
  }
  return dp[m][n];
};
```

Longest Increasing Subsequence

TypeScript

```
const lis = (nums: number[]): number => {
  if (nums.length === 0) return 0;
  const dp: number[] = new Array(nums.length).fill(1);
  let maxLength = 1;

  for (let i = 1; i < nums.length; i++) {
    for (let j = 0; j < i; j++) {
      if (nums[i] > nums[j]) {
        dp[i] = Math.max(dp[i], 1 + dp[j]);
      }
    }
    maxLength = Math.max(maxLength, dp[i]);
  }
  return maxLength;
};
```

Subset Sum

TypeScript

```typescript
const canPartition = (nums: number[], targetSum: number): boolean => {
  const n = nums.length;
  const dp: boolean[][] = Array(n + 1).fill(false).map(() => Array(targetSum + 1).fill(false));

  for (let i = 0; i <= n; i++) {
    dp[i][0] = true; // Base case: sum of 0 is always possible
  }

  for (let i = 1; i <= n; i++) {
    const num = nums[i - 1];
    for (let s = 1; s <= targetSum; s++) {
      if (s < num) {
        dp[i][s] = dp[i - 1][s]; // Exclude the current number
      } else {
        dp[i][s] = dp[i - 1][s] || dp[i - 1][s - num]; // Exclude or include
      }
    }
  }
  return dp[n][targetSum];
};
```

Matrix Chain Multiplication

TypeScript

```typescript
const matrixChainOrder = (p: number[]): number => {
  const n = p.length - 1;
  const m: number[][] = Array(n).fill(0).map(() => Array(n).fill(0));
```

```
  for (let len = 2; len <= n; len++) {
    for (let i = 0; i <= n - len; i++) {
      const j = i + len - 1;
      m[i][j] = Infinity;
      for (let k = i; k < j; k++) {
        const cost = m[i][k] + m[k + 1][j] + p[i] * p[k + 1] * p[j + 1];
        m[i][j] = Math.min(m[i][j], cost);
      }
    }
  }
  return m[0][n - 1];
};
```

---

2. Backtracking

TypeScript

```
type State = any; // Define your state type here

const solve = (state: State, results: any[]): void => {
  // Base case: is this a solution?
  if (isSolution(state)) {
    results.push(state.slice());
    return;
  }

  // Iterate through choices
  for (const choice of getPossibleChoices(state)) {
    // 1. Make a choice
    makeChoice(state, choice);

    // 2. Recurse
    solve(state, results);

    // 3. Backtrack (undo the choice)
    undoChoice(state, choice);
```

```
  }
};
```

---

3. Matrix Traversal

TypeScript

```
const directions = [[0, 1], [0, -1], [1, 0], [-1, 0]];

const isValid = (grid: number[][], r: number, c: number): boolean => {
  return r >= 0 && r < grid.length && c >= 0 && c < grid[0].length;
};

const traverseMatrix = (grid: number[][], startRow: number, startCol: number): void => {
  const visited: Set<string> = new Set();
  const queue: [number, number][] = [[startRow, startCol]];
  visited.add(`${startRow},${startCol}`);

  while (queue.length > 0) {
    const [row, col] = queue.shift()!;
    // Process grid[row][col]

    for (const [dr, dc] of directions) {
      const newRow = row + dr;
      const newCol = col + dc;
      if (isValid(grid, newRow, newCol) && !visited.has(`${newRow},${newCol}`)) {
        visited.add(`${newRow},${newCol}`);
        queue.push([newRow, newCol]);
      }
    }
  }
};
```

---

4. Breadth-First Search (BFS)

TypeScript

```
const bfs = (graph: Map<number, number[]>, startNode: number): void => {
  const queue: number[] = [startNode];
  const visited: Set<number> = new Set([startNode]);

  while (queue.length > 0) {
    const node = queue.shift()!;
    // Process `node`

    for (const neighbor of graph.get(node) || []) {
      if (!visited.has(neighbor)) {
        visited.add(neighbor);
        queue.push(neighbor);
      }
    }
  }
};
```

---

5. Depth-First Search (DFS)

TypeScript

```
// Recursive DFS
const dfsRecursive = (graph: Map<number, number[]>, node: number, visited: Set<number> = new Set()): void => {
  if (visited.has(node)) return;
  visited.add(node);
  // Process `node`
```

```
  for (const neighbor of graph.get(node) || []) {
    dfsRecursive(graph, neighbor, visited);
  }
};

// Iterative DFS
const dfsIterative = (graph: Map<number, number[]>, startNode: number): void => {
  const stack: number[] = [startNode];
  const visited: Set<number> = new Set();

  while (stack.length > 0) {
    const node = stack.pop()!;
    if (visited.has(node)) continue;
    visited.add(node);
    // Process `node`

    for (const neighbor of graph.get(node) || []) {
      stack.push(neighbor);
    }
  }
};
```

---

6. Binary Tree Traversal

TypeScript

```
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

// In-order traversal (Left -> Root -> Right)
const inorder = (node: TreeNode | null): void => {
  if (!node) return;
```

```
  inorder(node.left);
  console.log(node.val); // Process node
  inorder(node.right);
};

// Pre-order traversal (Root -> Left -> Right)
const preorder = (node: TreeNode | null): void => {
  if (!node) return;
  console.log(node.val); // Process node
  preorder(node.left);
  preorder(node.right);
};

// Post-order traversal (Left -> Right -> Root)
const postorder = (node: TreeNode | null): void => {
  if (!node) return;
  postorder(node.left);
  postorder(node.right);
  console.log(node.val); // Process node
};
```

---

7. Modified Binary Search

TypeScript

```
const binarySearch = (arr: number[], target: number): number => {
  let left = 0;
  let right = arr.length - 1;

  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);
    if (arr[mid] === target) return mid; // Found it
    if (arr[mid] < target) {
      left = mid + 1; // Search right half
    } else {
      right = mid - 1; // Search left half
```

```
    }
  }
  return -1; // Not found
};
```

---

8. Overlapping Interval Pattern

TypeScript

```
interface Interval {
  start: number;
  end: number;
}

const mergeIntervals = (intervals: Interval[]): Interval[] => {
  if (intervals.length < 2) return intervals;

  intervals.sort((a, b) => a.start - b.start);

  const merged: Interval[] = [intervals[0]];
  for (let i = 1; i < intervals.length; i++) {
    const last = merged[merged.length - 1];
    const current = intervals[i];

    if (current.start <= last.end) {
      // Overlap, merge them
      last.end = Math.max(last.end, current.end);
    } else {
      // No overlap, add new interval
      merged.push(current);
    }
  }
  return merged;
};
```

---

9. Top K Elements Pattern

TypeScript

```
// Assuming a MinHeap class exists
class MinHeap {
  // ... implementation ...
  size(): number { return 0; }
  enqueue(val: number): void {}
  dequeue(): number { return 0; }
  peek(): number { return 0; }
}

const findTopKElements = (nums: number[], k: number): number[] => {
  const minHeap = new MinHeap(); // Use a real MinHeap here

  for (const num of nums) {
    minHeap.enqueue(num);
    if (minHeap.size() > k) {
      minHeap.dequeue();
    }
  }

  // Convert heap to array for result
  const result: number[] = [];
  while (minHeap.size() > 0) {
    result.push(minHeap.dequeue());
  }
  return result;
};
```

---

10. Monotonic Stack

TypeScript

```typescript
const nextGreaterElement = (nums: number[]): number[] => {
  const stack: number[] = []; // Stores indices
  const result: number[] = new Array(nums.length).fill(-1);

  for (let i = 0; i < nums.length; i++) {
    // Pop elements from stack that are smaller than current num
    while (stack.length > 0 && nums[stack[stack.length - 1]] < nums[i]) {
      const prevIndex = stack.pop()!;
      result[prevIndex] = nums[i];
    }
    stack.push(i);
  }
  return result;
};
```

---

11. Linked List In-Place Reversal

TypeScript

```typescript
interface ListNode {
  val: number;
  next: ListNode | null;
}

const reverseList = (head: ListNode | null): ListNode | null => {
  let prev: ListNode | null = null;
  let current: ListNode | null = head;

  while (current !== null) {
```

```
    const nextTemp: ListNode | null = current.next;
    current.next = prev; // Reverse the link
    prev = current;      // Move prev to current
    current = nextTemp;  // Move current to next
  }
  return prev;
};
```

---

12. Fast and Slow Pointers

TypeScript

```
interface ListNode {
  val: number;
  next: ListNode | null;
}

const hasCycle = (head: ListNode | null): boolean => {
  let slow: ListNode | null = head;
  let fast: ListNode | null = head;

  while (fast !== null && fast.next !== null) {
    slow = slow!.next;
    fast = fast.next.next;
    if (slow === fast) {
      return true; // Cycle detected
    }
  }
  return false;
};
```

---

13. Sliding Window Pattern

TypeScript

```typescript
const findMaxSumSubarray = (arr: number[], k: number): number => {
  let windowSum = 0;
  let windowStart = 0;
  let maxSum = 0;

  for (let windowEnd = 0; windowEnd < arr.length; windowEnd++) {
    windowSum += arr[windowEnd]; // Grow window

    // When window reaches size k, check sum and shrink
    if (windowEnd >= k - 1) {
      maxSum = Math.max(maxSum, windowSum);
      windowSum -= arr[windowStart]; // Shrink window from the left
      windowStart++;
    }
  }
  return maxSum;
};
```

---

14. Two Pointers

TypeScript

```typescript
const findPairSum = (arr: number[], target: number): [number, number] | [-1, -1] => {
  let left = 0;
  let right = arr.length - 1;

  while (left < right) {
    const currentSum = arr[left] + arr[right];
    if (currentSum === target) {
```

```
      return [left, right];
    } else if (currentSum < target) {
      left++; // Need a larger sum
    } else {
      right--; // Need a smaller sum
    }
  }
  return [-1, -1];
};
```

---

15. Prefix Sum

TypeScript

```
// Create a prefix sum array
const createPrefixSum = (nums: number[]): number[] => {
  const prefixSum: number[] = [0];
  for (const num of nums) {
    prefixSum.push(prefixSum[prefixSum.length - 1] + num);
  }
  return prefixSum;
};

// Use the prefix sum array
// sumRange(prefixSum, 0, 2) would be nums[0] + nums[1] + nums[2]
const sumRange = (prefixSum: number[], left: number, right: number): number => {
  return prefixSum[right + 1] - prefixSum[left];
};
```