JS

Top 10 Algorithms in JavaScript

Bubble Sort

A simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

```
function bubbleSort(arr) {
  let n = arr.length;
  for (let i = 0; i < n - 1; i++) {
    for (let j = 0; j < n - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        // Swap
        let temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
  return arr;
}
```

Quick Sort

A fast sorting algorithm that uses a divide-and-conquer strategy to sort elements.

```
function quickSort(arr) {
  if (arr.length <= 1) return arr;

  const pivot = arr[0];
  const left = [];
  const right = [];

  for (let i = 1; i < arr.length; i++) {
    arr[i] < pivot ? left.push(arr[i]) : right.push(arr[i]);
  }

  return quickSort(left).concat(pivot, quickSort(right));
}
```

Binary Search

An efficient search algorithm that finds the position of a target value within a sorted array.

```
function binarySearch(arr, target) {
  let low = 0;
  let high = arr.length - 1;

  while (low <= high) {
    const mid = Math.floor((low + high) / 2);
    const guess = arr[mid];

    if (guess === target) return mid;
    if (guess > target) high = mid - 1;
    else low = mid + 1;
  }

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

Linear Search

A simple search algorithm that finds the position of a target value within a list.

```
function linearSearch(arr, target) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      return i;
    }
  }
  return -1; // Not found
}
```

Knapsack Problem

A classic optimization problem where the goal is to select items in a way that maximizes the total value without exceeding a given weight.

```
function knapsack(weights, values, capacity) {
  const n = weights.length;
  const dp = Array.from({ length: n + 1 }, () => Array(capacity + 1).fill(0));

  for (let i = 1; i <= n; i++) {
    for (let w = 1; w <= capacity; w++) {
      if (weights[i - 1] <= w) {
        dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
      } else {
        dp[i][w] = dp[i - 1][w];
      }
    }
  }

  return dp[n][capacity];
}
```

Fibonacci Sequence

A series of numbers where each number is the sum of the two preceding ones, often used as an example for dynamic programming.

```
function fibonacci(n) {
  const fib = [0, 1];

  for (let i = 2; i <= n; i++) {
    fib[i] = fib[i - 1] + fib[i - 2];
  }

  return fib[n];
}
```

Merge Sort

Another efficient sorting algorithm that uses a divide-and-conquer approach.

```
function mergeSort(arr) {
  if (arr.length <= 1) return arr;

  const middle = Math.floor(arr.length / 2);
  const left = arr.slice(0, middle);
  const right = arr.slice(middle);

  return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right) {
  let result = [];
  let leftIndex = 0;
  let rightIndex = 0;

  while (leftIndex < left.length && rightIndex < right.length) {
    if (left[leftIndex] < right[rightIndex]) {
      result.push(left[leftIndex]);
      leftIndex++;
    } else {
      result.push(right[rightIndex]);
      rightIndex++;
    }
  }

  return result.concat(left.slice(leftIndex), right.slice(rightIndex));
}
```

Depth-First Search (DFS)

A traversal algorithm that explores as far as possible along each branch before backtracking.

```
class Graph {
  constructor() {
    this.adjList = new Map();
  }

  addVertex(vertex) {
    this.adjList.set(vertex, []);
  }

  addEdge(v, w) {
    this.adjList.get(v).push(w);
    this.adjList.get(w).push(v);
  }

  dfs(startingNode) {
    const visited = new Set();

    function dfsHelper(node) {
      visited.add(node);
      console.log(node);

      const neighbors = this.adjList.get(node);
      for (const neighbor of neighbors) {
        if (!visited.has(neighbor)) {
          dfsHelper(neighbor);
        }
      }
    }

    dfsHelper(startingNode);
  }
}
```

Dijkstra's Algorithm

A graph search algorithm that finds the shortest path between two nodes in a weighted graph.

```
function dijkstra(graph, start) {
  const distances = {};
  const visited = new Set();

  for (const vertex in graph) {
    distances[vertex] = Infinity;
  }

  distances[start] = 0;

  while (true) {
    const current = getMinNode(distances, visited);
    if (!current) break;

    visited.add(current);

    for (const neighbor in graph[current]) {
      const newDistance = distances[current] + graph[current][neighbor];
      if (newDistance < distances[neighbor]) {
        distances[neighbor] = newDistance;
      }
    }
  }

  return distances;
}

function getMinNode(distances, visited) {
  return Object.keys(distances).reduce((min, node) => {
    if (!visited.has(node) && distances[node] < distances[min]) {
      return node;
    }
    return min;
  }, null);
}
```

Breadth-First Search (BFS)

Another traversal algorithm that visits all the vertices of a graph in breadthward motion.

```
function bfs(graph, start) {
  const visited = new Set();
  const queue = [start];

  while (queue.length > 0) {
    const node = queue.shift();

    if (!visited.has(node)) {
      console.log(node);
      visited.add(node);

      const neighbors = graph[node];
      for (const neighbor of neighbors) {
        queue.push(neighbor);
      }
    }
  }
}
```

Conclusion

These algorithms cover a range of concepts including sorting, searching, graph traversal, dynamic programming, and optimization. Understanding these algorithms will provide a strong foundation for solving various programming challenges.