Zohaib Asghar

JavaScript Data Structures & Algorithms

Essential Cheat Sheet

```
class Node {
  constructor(value) {
    this.value = value;
    this.next = null;
  }
}
```

CODINGWITHJD

Swipe for more >>

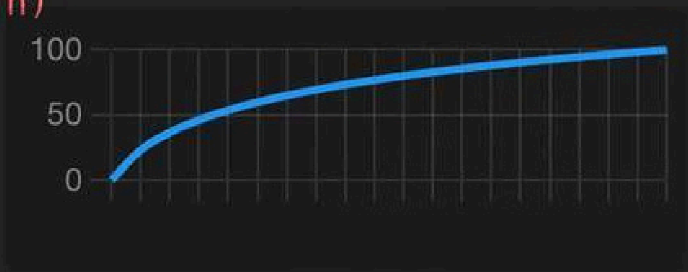
Big O Notation

Time Complexity Analysis

$O(1)$ — Constant Time

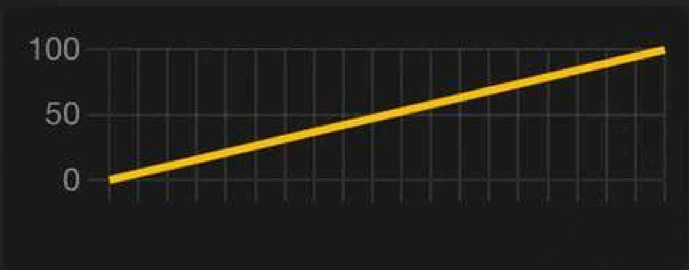
Always takes same time regardless of input size. Like accessing array index.

$O(\log n)$ — Logarithmic Time



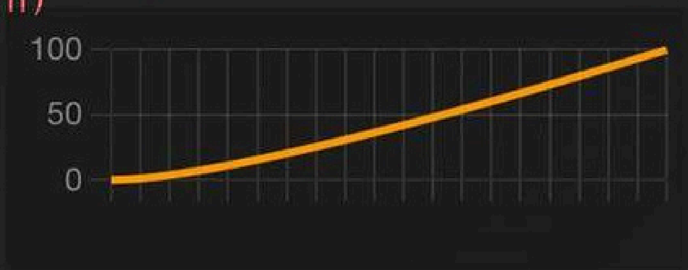
Cuts problem in half each time. Like binary search.

$O(n)$ — Linear Time



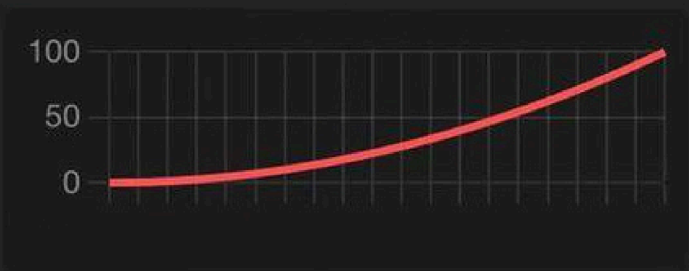
Time grows linearly with input. Like linear search.

$O(n \log n)$ — Linearithmic Time



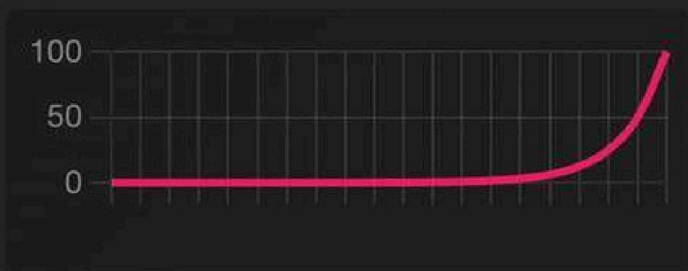
Common in efficient sorting algorithms like merge sort.

$O(n^2)$ — Quadratic Time



Nested iterations. Like bubble sort.

$O(2^n)$ — Exponential Time



Solutions double with each addition. Like recursive fibonacci.

CODINGWITHJD

Arrays

Common Operations & Time Complexity

Access

O(1)

Get or set element directly using index position. Fastest array operation.

```
const value = array[5];
```

Push/Pop

O(1)

Add (push) or remove (pop) element at end. Efficient as no reindexing needed.

```
array.push(1); // add
array.pop();    // remove
```

Unshift/Shift

O(n)

Add (unshift) or remove (shift) at start. Requires reindexing all elements.

```
array.unshift(1); // add
array.shift();     // remove
```

Search

O(n)

Find element by value. Must check each element until match found.

```
array.indexOf(5);
array.find(x => x > 3);
```

Insert

O(n)

Insert at position. Requires shifting all subsequent elements right.

```
array.splice(2, 0, 5);
```

Delete

O(n)

Remove at position. Requires shifting all subsequent elements left.

```
array.splice(2, 1);
```

Linked List

Linear Data Structure

Insert Head — O(1)

Add new node at beginning. Just update head pointer.

```
node.next = head;
head = node;
```

Insert Tail — O(1)*

Add at end with tail pointer. O(n) without tail.

```
tail.next = node;
tail = node;
```

Delete Head — O(1)

Remove first node. Update head pointer.

```
head = head.next;
```

Insert/Delete Middle — O(n)

Need to traverse to find position first.

```
prev.next = node;
node.next = current;
```

Search — O(n)

Must traverse nodes sequentially.

```
while(current) {
  if(current.val === x) return true;
  current = current.next;
}
```

Access by Index — O(n)

No direct access. Must traverse from start.

```
for(let i = 0; i < index; i++) {
  current = current.next;
}
```

Binary Trees

Common Operations

Insert

O(log n)*

Add new node in BST. O(n) worst case for unbalanced.

```
if (value < node.val)
  node.left = insert(node.left, value);
else
  node.right = insert(node.right, value);
```

Search

O(log n)*

Find node in BST. O(n) worst case for unbalanced.

```
if (!node || node.val === value) return node;
return value < node.val
  ? search(node.left, value)
  : search(node.right, value);
```

DFS Traversal

O(n)

Depth-first: preorder, inorder, postorder.

```
// Inorder (sorted in BST)
inorder(node) {
  inorder(node.left);
  visit(node);
  inorder(node.right);
}
```

BFS Traversal

O(n)

Level by level using queue.

```
const queue = [root];
while (queue.length) {
  const node = queue.shift();
  queue.push(node.left, node.right);
}
```

Delete

O(log n)*

Remove node and maintain BST property.

```
// Find min in right subtree
// Replace node with min
// Delete min from right
```

Height/Depth

O(n)

Find max depth of tree.

```
return !node ? 0 : 1 + Math.max(
  height(node.left),
  height(node.right)
);
```

Graphs

Common Operations

Add Vertex

O(1)

Add new vertex to adjacency list.

```
graph[vertex] = [];
```

Add Edge

O(1)

Connect two vertices (nodes).

```
graph[v1].push(v2);
// For undirected:
graph[v2].push(v1);
```

BFS

O(V + E)

Breadth-first traversal using queue.

```
const q = [start];
while (q.length) {
  const v = q.shift();
  q.push(...graph[v]);
}
```

DFS

O(V + E)

Depth-first traversal using recursion/stack.

```
function dfs(vertex) {
  visited.add(vertex);
  graph[vertex].forEach(v => {
    if (!visited.has(v)) dfs(v);
  });
}
```

Find Path

O(V + E)

Find path between two vertices.

```
// Using BFS/DFS
while (queue.length) {
  const v = queue.shift();
  if (v === target) return path;
}
```

Check Cycle

O(V + E)

Detect cycle in directed/undirected graph.

```
function hasCycle(v, parent) {
  if (visited.has(v)) return true;
  return graph[v].some(next =>
    next !== parent && hasCycle(next, v)
  );
}
```

Sorting

Common Algorithms

Bubble Sort

$O(n^2)$

Compare adjacent elements and swap if needed. Simple but inefficient.

```
for (let i = 0; i < n; i++) {
  for (let j = 0; j < n - i - 1; j++) {
    if (arr[j] > arr[j + 1]) {
      [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
    }
  }
}
```

Quick Sort

$O(n \log n)$*

Pick pivot, partition array, recursively sort. $O(n^2)$ worst case.

```
function quickSort(arr, low, high) {
  if (low < high) {
    let pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}
```

Merge Sort

$O(n \log n)$

Divide array, sort halves, merge results. Stable sort.

```
function mergeSort(arr) {
  if (arr.length <= 1) return arr;
  let mid = Math.floor(arr.length / 2);
  return merge(
    mergeSort(arr.slice(0, mid)),
    mergeSort(arr.slice(mid))
  );
}
```

Insertion Sort

$O(n^2)$

Build sorted array one item at a time. Good for small n.

```
for (let i = 1; i < n; i++) {
  let key = arr[i];
  let j = i - 1;
  while (j >= 0 && arr[j] > key) {
    arr[j + 1] = arr[j];
    j--;
  }
  arr[j + 1] = key;
}
```

Heap Sort

$O(n \log n)$

Build max heap, repeatedly extract max. In-place sort.

```
function heapSort(arr) {
  buildMaxHeap(arr);
  for (let i = arr.length - 1; i > 0; i--) {
    [arr[0], arr[i]] = [arr[i], arr[0]];
    heapify(arr, 0, i);
  }
}
```

Selection Sort

$O(n^2)$

Find min element, place at start. Simple but inefficient.

```
for (let i = 0; i < n - 1; i++) {
  let min = i;
  for (let j = i + 1; j < n; j++) {
    if (arr[j] < arr[min]) min = j;
  }
  if (min !== i) [arr[i], arr[min]] = [arr[min], arr[i]];
}
```

Search

Common Algorithms

Linear Search

O(n)

Check each element sequentially. Works on unsorted arrays.

```
function linearSearch(arr, target) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) return i;
  }
  return -1;
}
```

Binary Search

O(log n)

Divide and conquer. Requires sorted array.

```
function binarySearch(arr, target) {
  let left = 0, right = arr.length - 1;
  while (left <= right) {
    let mid = Math.floor((left + right) / 2);
    if (arr[mid] === target) return mid;
    if (arr[mid] < target) left = mid + 1;
    else right = mid - 1;
  }
  return -1;
}
```

Jump Search

O(√n)

Jump fixed steps, then linear search. For sorted arrays.

```
function jumpSearch(arr, target) {
  const step = Math.floor(Math.sqrt(arr.length));
  let prev = 0;
  while (arr[Math.min(step, arr.length) - 1] < target) {
    prev = step;
    step += Math.floor(Math.sqrt(arr.length));
    if (prev >= arr.length) return -1;
  }
  // Linear search in block
  while (arr[prev] < target) {
    prev++;
    if (prev === Math.min(step, arr.length)) return -1;
  }
  return arr[prev] === target ? prev : -1;
}
```

Interpolation Search

O(log log n)*

Like binary search but with better position guessing.

```
function interpolationSearch(arr, target) {
  let low = 0, high = arr.length - 1;
  while (low <= high && target >= arr[low]
         && target <= arr[high]) {
    let pos = low + Math.floor(
      (target - arr[low]) * (high - low) /
      (arr[high] - arr[low])
    );
    if (arr[pos] === target) return pos;
    if (arr[pos] < target) low = pos + 1;
    else high = pos - 1;
  }
  return -1;
}
```

REPOST & FOLLOW **ZOHAIB ASGHAR** FOR MORE CONTENT LIKE THIS.

Zohaib Asghar

@zohaibAsghar