Big O Common Code Patterns

O(1) | O(log n) | O(n) | O(n log n) | O(n²) | O(2ⁿ)

Array Access

Direct access of an element by index

```
let arr = [1, 2, 3, 4, 5]
let firstElement = arr[0] // O(1)
let lastElement = arr[arr.length - 1] // O(1)
```

Common Use Cases: Quick lookups in fixed-size arrays

Simple Computation

Sum two integers

```
func addOne(n: Int) {
    return n + 1
}
```

Common Use Cases: Managing function calls, undo operations

O(1) | O(log n) | O(n) | O(n log n) | O(n²) | O(2ⁿ)

Binary Search

Halving the search space in each step

```
func binarySearch(_ arr: [Int], _ target: Int) -> Int {
    var left = 0, right = arr.count - 1

    while left <= right {
        let mid = (left + right) / 2
        if arr[mid] == target { return mid }
        if arr[mid] < target {
            left = mid + 1
        } else {
            right = mid - 1
        }
    }
    return -1
}
```

Common Use Cases: Finding elements in sorted arrays

Height-Balanced BST Operations

Operations on a balanced binary search tree

```
class AVLTree {
    func insert(_ value: Int) {
        // O(log n) - always balanced
        root = insert(root, value)
        balance(root)
    }

    func search(_ value: Int) -> Bool {
        // O(log n) - tree height
        return search(root, value)
    }
}
```

Common Use Cases: Maintaining sorted data with fast operations

O(1) | O(log n) | O(n) | O(n log n) | O(n²) | O(2ⁿ)

Simple Iteration

Processing each elements of a collection

```
func sum(_ arr: [Int]) {
    var sum = 0
    for i in 0..<arr.count {
        sum += arr[i]
    }
    return sum
}
```

Common Use Cases: Key-value storage with potential collisions

Array/String Search

Finding an element in an unsorted array

```
func linearSearch(_ arr: [Int], _ target: Int) -> Int {
    for (index, value) in arr.enumerated() {
        if value == target {
            return index
        }
    }
    return -1
}
```

Common Use Cases: Finding elements in unsorted data

O(1) | O(log n) | O(n) | O(n log n) | O(n²) | O(2ⁿ)

Merge Sort

Divide and conquer sorting algorithm

```
func mergeSort(_ arr: [Int]) -> [Int] {
    guard arr.count > 1 else { return arr }

    let mid = arr.count / 2
    let left = mergeSort(Array(arr[..<mid]))
    let right = mergeSort(Array(arr[mid...]))
    return merge(left, right)
}
```

Common Use Cases: Guaranteed efficient sorting of arrays

Stable Sort

Built-in sorting with stability guarantee

```
let items = ["abc", "def", "ghi"]
let sorted = items.sorted { $0 < $1 }
```

Common Use Cases: General-purpose sorting when stability matters

O(1) | O(log n) | O(n) | O(n log n) | O(n²) | O(2ⁿ)

Nested Iteration

Processing each element against every other

```
func findAllPairs(_ arr: [Int]) {
    for i in 0..<arr.count {
        for j in i+1..<arr.count {
            if arr[i] + arr[j] == 10 {
                print("Pair: \(arr[i]), \(arr[j])")
            }
        }
    }
}
```

Common Use Cases: Finding all possible combinations of pairs

Quick Sort (Worst Case)

When pivot selection is unfortunate

```
func quickSort(_ arr: inout [Int], _ low: Int, _ high: Int) {
    // O(n²) when array is already sorted
    // or reverse sorted
    guard low < high else { return }

    let pivot = partition(&arr, low, high)
    quickSort(&arr, low, pivot - 1)
    quickSort(&arr, pivot + 1, high)
}
```

Common Use Cases: General sorting when worst case is acceptable

O(1) | O(log n) | O(n) | O(n log n) | O(n²) | O(2ⁿ)

Recursive Fibonacci

Naive recursive implementation

```
func fibonacci(_ n: Int) -> Int {
    // Each call spawns two more calls
    if n <= 1 { return n }
    return fibonacci(n - 1) + fibonacci(n - 2)
}
```

Common Use Cases: Educational purposes (not for production)

Power Set

Generate all possible subsets

```
func generateSubsets(_ arr: [Int]) -> [[Int]] {
    // 2^ subsets for n elements
    if arr.isEmpty { return [[]] }
    let first = arr[0]
    let subsetsWithoutFirst = generateSubsets(
        Array(arr.dropFirst())
    )
    let subsetsWithFirst = subsetsWithoutFirst.map {
        [first] + $0
    }
    return subsetsWithoutFirst + subsetsWithFirst
}
```

Common Use Cases: When all combinations must be considered