



DSA in Python & JS

Big-O · patterns · recursion · searching · sorting · strings

Python

JavaScript

Notes by Sandeep Kumar Jha · Senior Full Stack Developer

Revision notes · handwritten style · based on Technical Suneja's "DSA in JavaScript" playlist (#1-#21)

Every solution written twice — Python + JavaScript — and tested

Part of my interview-revision series

Contents

- 1 · How to approach DSA + Big-O
- 2 · The 4 problem-solving patterns
- 3 · Sum Zero — brute → two pointers (#1, #2)
- 4 · Valid Anagram — frequency counter (#3)
- 5 · Count Unique Values — two pointers (#4)
- 6 · Largest Sum of Consecutive Elements — sliding window (#5)
- 7 · Divide & Conquer — Binary Search (#6)
- 8 · Squares in Another Array (#7, #8)
- 9 · Recursion basics (#9)
- 10 · Advanced recursion (#10)
- 11 · Helper-method recursion (#12)
- 12 · Searching — Linear vs Binary (#13, #14)
- 13 · Bubble Sort + optimisation (#15, #16)
- 14 · Selection Sort (#17)
- 15 · Insertion Sort + sorting summary (#18)
- 16 · Maximum Occurring Character (#19)
- 17 · Palindrome Check (#20)
- 18 · Longest Substring Without Repeating Chars (#21)
- 19 · Python ↔ JS cheat sheet · Rapid-fire Q&A

How to use these notes

-  Read the problem, **cover the code**, solve it on paper in one language, then write it in the other.
-  Yellow = remember. Orange box = revision / interview. Red = common mistake. Purple = definition / problem statement.
-  `python` and `javascript` cards are side by side. All code is copied from `exercises/dsa/` where it is tested.
-  "#n" = video number in the Technical Suneja playlist (the videos teach in JavaScript; the Python versions are my own translation).

1

How to approach DSA + Big-O

Definition

Data structure = a way to store & organise data (array, object/dict, set, stack...). **Algorithm** = a step-by-step recipe to solve a problem. DSA = picking the right structure + recipe so the solution is **correct and fast**.

My 5-step problem-solving routine

1. **Understand** — restate the problem; inputs? outputs? sorted? negatives? empty input?
2. **Examples** — simple case, edge cases ([], one item, duplicates, no answer).
3. **Brute force first** — any working solution, say its Big-O out loud.
4. **Optimise** — look for a pattern: frequency counter, two pointers, sliding window, divide & conquer.
5. **Refactor & test** — clean names, re-run the examples, state final time + space.

Big-O = how work grows with input size n

python

```
def add_upto_slow(n):          # O(n)
    total = 0
    for i in range(1, n + 1):
        total += i
    return total
```

javascript

```
function addUptoSlow(n) {      // O(n)
    let total = 0;
    for (let i = 1; i <= n; i++) total += i;
    return total;
}
```

python

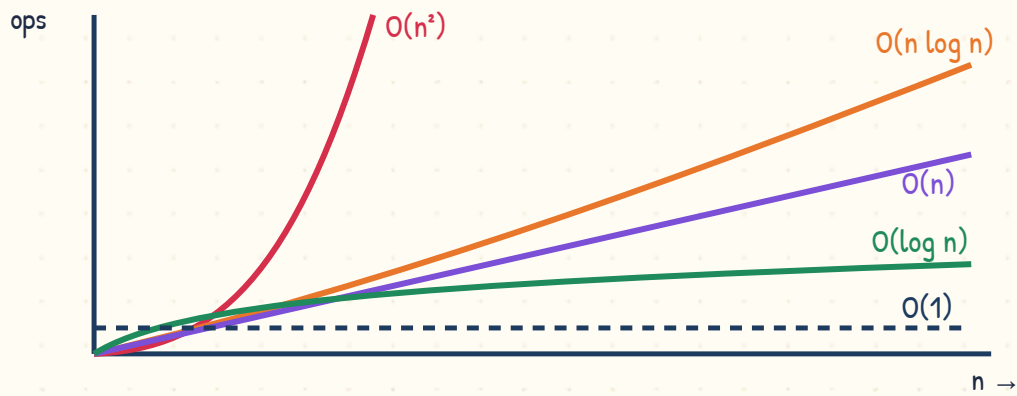
```
def add_upto_fast(n):          # O(1)
    return n * (n + 1) // 2
```

javascript

```
function addUptoFast(n) {      // O(1)
    return (n * (n + 1)) / 2;
}
```

same answer — but one does n additions, the other does 3 operations

Big-O	Name	Example	$n = 1,000$
$O(1)$	Constant	Array index <code>arr[5]</code> , dict/object lookup, formula	1
$O(\log n)$	Logarithmic	Binary search — halve each step	~10
$O(n)$	Linear	One loop, linear search, frequency counter	1,000
$O(n \log n)$	Linearithmic	Merge sort, built-in <code>sort()</code>	~10,000
$O(n^2)$	Quadratic	Nested loops, bubble / selection / insertion sort	1,000,000
$O(2^n)$	Exponential	Naive recursive Fibonacci	astronomical



Big-O rules

- Drop constants:** $O(2n) \rightarrow O(n)$. $O(500) \rightarrow O(1)$.
- Keep the biggest term:** $O(n^2 + n) \rightarrow O(n^2)$.
- Loops one after another **add** ($O(n + n) = O(n)$); loops **inside** each other **multiply** ($O(n \cdot n) = O(n^2)$).
- We talk about the **worst case** unless told otherwise.
- Space complexity** = extra memory the algorithm uses (a new dict of n keys $\rightarrow O(n)$; a few variables $\rightarrow O(1)$).

Python $\rightleftharpoons$ JS – the structures we use

Idea	Python	JavaScript	Cost
Dynamic array	<code>list</code> $\rightarrow$ <code>[1, 2]</code>	<code>Array</code> $\rightarrow$ <code>[1, 2]</code>	index / push $O(1)$, insert front $O(n)$
Key $\rightarrow$ value	<code>dict</code> $\rightarrow$ <code>{"a": 1}</code>	<code>{}</code> object or <code>new Map()</code>	get / set $O(1)$ avg
Unique values	<code>set()</code>	<code>new Set()</code>	add / has $O(1)$ avg
String	<code>str</code> (immutable)	<code>string</code> (immutable)	slice $O(k)$
Integer divide	<code>a // b</code>	<code>Math.floor(a / b)</code>	

Revision

Understand $\rightarrow$ examples $\rightarrow$ brute force $\rightarrow$ optimise with a pattern $\rightarrow$ refactor. Big-O: drop constants, keep the biggest term, nested loops multiply. Always state **time AND space**.

2

The 4 problem-solving patterns

Why patterns?

Most beginner/intermediate array & string questions are solved by turning an $O(n^2)$ nested loop into $O(n)$ using one of these 4 tricks. The whole playlist is built around them.

Pattern	Idea	Signal in the question	Used in
Frequency counter	Count things in a dict/object, then compare counts instead of nested loops.	"same letters", "same frequency", "most common", compare two collections	Anagram, Squares, Max char
Multiple / two pointers	Two indexes moving towards each other or in the same direction based on a condition.	Sorted array, pairs, "in place", palindrome	Sum Zero, Count Unique, Palindrome
Sliding window	Keep a window (start...end) over the data; slide it instead of recomputing.	"consecutive", "subarray / substring", "longest / max of size k"	Largest sum, Longest substring
Divide & conquer	Split the input into halves, solve the smaller part, discard the rest.	Sorted data, "find" fast, $O(\log n)$ hinted	Binary search

Frequency counter shape

```
freq = {}
for x in data:
    freq[x] = freq.get(x, 0) + 1
# compare / scan freq → O(n)
```

Two pointers shape

```
left, right = 0, len(arr) - 1
while left < right:
    # decide: left += 1 or right -= 1
```

Sliding window shape

```
window = sum(arr[:k])
for i in range(k, len(arr)):
    window += arr[i] - arr[i-k]
```

Divide & conquer shape

```
lo, hi = 0, len(arr) - 1
while lo <= hi:
    mid = (lo + hi) // 2
    # keep one half
```

Revision

Compare/count → frequency counter. Sorted + pairs → two pointers. Consecutive / substring → sliding window. Sorted + find fast → divide & conquer.

Problem

Given a **sorted** array of integers, return the **first pair** whose sum is 0. If no pair, return **None** / **undefined**.

`[-4, -3, -2, -1, 0, 1, 2, 5] → [-2, 2] · [-3, -2, -1, 0, 5] → none`

Way 1 · Brute force – $O(n^2)$

Check every pair with two nested loops.

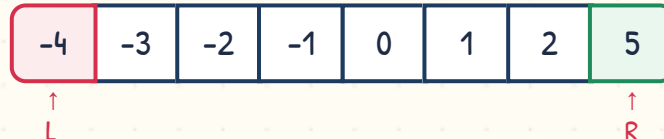
python

```
def sum_zero_brute(arr):
    # O(n^2)
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] + arr[j] == 0:
                return [arr[i], arr[j]]
    return None
```

javascript

```
function sumZeroBrute(arr) {
    // O(n^2)
    for (let i = 0; i < arr.length; i++) {
        for (let j = i + 1; j < arr.length; j++) {
            if (arr[i] + arr[j] === 0) {
                return [arr[i], arr[j]];
            }
        }
    }
    return undefined;
}
```

Way 2 · Two pointers – $O(n)$ time, $O(1)$ space



$-4 + 5 = 1 > 0 \rightarrow$ too big $\rightarrow$ move R left. If sum $< 0 \rightarrow$ too small $\rightarrow$ move L right.

$-4 + 5 = 1 > 0 \rightarrow R--$

$-4 + 2 = -2 < 0 \rightarrow L++$

$-3 + 2 = -1 < 0 \rightarrow L++$

$-2 + 2 = 0 \quad \checkmark$ return `[-2, 2]`

python

```
def sum_zero(arr):
    # sorted input
    left, right = 0, len(arr) - 1
    # O(n), O(1)
    while left < right:
        total = arr[left] + arr[right]
        if total == 0:
            return [arr[left], arr[right]]
        elif total > 0:
            right -= 1
            # need smaller
        else:
            left += 1
            # need bigger
    return None
```

javascript

```
function sumZero(arr) {
    // sorted input
    let left = 0;
    let right = arr.length - 1;
    while (left < right) {
        const sum = arr[left] + arr[right];
        if (sum === 0) return [arr[left], arr[right]];
        else if (sum > 0) right--;
        // need smaller
        else left++;
        // need bigger
    }
    return undefined;
}
```

Mistake

Using `left <= right` → when they meet you add the **same element** to itself ($0 + 0 = 0$ gives a wrong pair `[0, 0]`). Use `left < right`. And two pointers only work because the array is **sorted**.

Revision

Sorted + find a pair → two pointers from both ends. Sum too big → `right--`, too small → `left++`. $O(n^2)$ → $O(n)$.

Problem

Two strings are **anagrams** if they use **exactly the same letters, the same number of times**, in any order.

"anagram", "nagaram" → true · "rat", "car" → false · "aaz", "zza" → false

Approach	How	Big-O
Nested loops	For each char of a, search & remove it from b	$O(n^2)$
Sort & compare	<code>sorted(a) == sorted(b)</code>	$O(n \log n)$
Frequency counter ✓	Count chars of a, then "use them up" with b	$O(n)$ time, $O(k)$ space

a:3 n:1 g:1 r:1 m:1

freq after counting "anagram" → each char of "nagaram" decrements; any missing / zero count → false

python

```
def is_anagram(a, b):
    # O(n)
    if len(a) != len(b):
        return False
    freq = {}
    for ch in a:
        # count a
        freq[ch] = freq.get(ch, 0) + 1
    for ch in b:
        # use up with b
        if not freq.get(ch):
            return False
        freq[ch] -= 1
    return True

# one-liner: Counter(a) == Counter(b)
```

javascript

```
function isAnagram(a, b) {
    // O(n)
    if (a.length !== b.length) return false;
    const freq = {};
    for (const ch of a) {
        // count a
        freq[ch] = (freq[ch] || 0) + 1;
    }
    for (const ch of b) {
        // use up with b
        if (!freq[ch]) return false;
        freq[ch]--;
    }
    return true;
}
```

Python: `Counter(a) == Counter(b)`. JS has no Counter → memorise `freq[ch] = (freq[ch] || 0) + 1`.

Mistake

Forgetting the **length check** first, and checking only "does b's char exist in a" (that says "aaz" and "zza" match). Counts matter, not just presence. Ask the interviewer about **case & spaces** ("Dormitory" vs "dirty room").

Revision

Anagram = same chars, same counts. Length check → count first string → decrement with second → any miss = false. $O(n)$.

Problem

Given a **sorted** array, count how many **unique** values it has. Can be negative.

`[1,1,1,2,3,3,4,4,5,6]` → 6 · `[]` → 0

Two pointers, same direction

i = last unique value found, j = scout scanning ahead. When j finds a new value, move i one step and copy the value there.



python

```
def count_unique(arr):
    # sorted input
    if not arr:
        # 0(n), 0(1)
        return 0
    i = 0
    # last unique
    for j in range(1, len(arr)):
        if arr[j] != arr[i]:
            i += 1
            arr[i] = arr[j]
            # move it forward
    return i + 1

# easy way (0(n) extra space): len(set(arr))
```

javascript

```
function countUnique(arr) {
    // sorted input
    if (arr.length === 0) return 0; // 0(n), 0(1)
    let i = 0;
    // last unique
    for (let j = 1; j < arr.length; j++) {
        if (arr[j] !== arr[i]) {
            i++;
            arr[i] = arr[j];
            // move it forward
        }
    }
    return i + 1;
}

// easy way (0(n) extra space): new Set(arr).size
```

Mistake

Returning `i` instead of `i + 1` (i is an index), and crashing on an empty array. This version **modifies the input** – say so in an interview, or use a counter instead of copying.

Revision

Sorted → duplicates are neighbours. i = last unique, j scans; new value → $i++$ and copy. Answer $i + 1$. $O(n)$ time, $O(1)$ space. Shortcut: `len(set(arr))` / `new Set(arr).size` ($O(n)$ space).

Largest Sum of Consecutive Elements

Problem

Given an array and a number k , find the maximum sum of k consecutive elements.

$[2, 6, 9, 2, 1, 8, 5, 6, 3]$, $k=3 \rightarrow 19 (8 + 5 + 6)$

Way 1 · Brute force – $O(n \cdot k)$

For every starting index, add up the next k numbers again from scratch.

python

```
def max_sum_brute(arr, k):          #  $O(n \cdot k)$ 
    if len(arr) < k:
        return None
    best = float('-inf')
    for i in range(len(arr) - k + 1):
        best = max(best, sum(arr[i:i + k]))
    return best
```

javascript

```
function maxSumBrute(arr, k) {      //  $O(n \cdot k)$ 
    if (arr.length < k) return null;
    let best = -Infinity;
    for (let i = 0; i <= arr.length - k; i++) {
        let sum = 0;
        for (let j = 0; j < k; j++) sum += arr[i + j];
        best = Math.max(best, sum);
    }
    return best;
}
```

Way 2 · Sliding window – $O(n)$

window 1 = 17

2	6	9	2	1	8	5	6	3
---	---	---	---	---	---	---	---	---

17 - 2 + 2 = 17

2	6	9	2	1	8	5	6	3
---	---	---	---	---	---	---	---	---

17 - 6 + 1 = 12

2	6	9	2	1	8	5	6	3
---	---	---	---	---	---	---	---	---

... best = 19

2	6	9	2	1	8	5	6	3
---	---	---	---	---	---	---	---	---

new window = old window - element leaving + element entering

python

```
def max_sum(arr, k): # O(n)
    if len(arr) < k:
        return None
    window = sum(arr[:k]) # first window
    best = window
    for i in range(k, len(arr)):
        window += arr[i] - arr[i - k] # slide
        best = max(best, window)
    return best
```

javascript

```
function maxSum(arr, k) { // O(n)
    if (arr.length < k) return null;
    let window = 0;
    for (let i = 0; i < k; i++) window += arr[i];
    let best = window; // first window
    for (let i = k; i < arr.length; i++) {
        window += arr[i] - arr[i - k]; // slide
        best = Math.max(best, window);
    }
    return best;
}
```

Mistake

Starting `best = 0` — wrong when all numbers are negative (`[-3, -1, -2], k=2` → `-3`). Start from the first window (or $-\infty$). And handle `len(arr) < k`.

Revision

"k consecutive" → fixed sliding window. Sum first k, then slide: `+ arr[i] - arr[i-k]`, track max. $O(n \cdot k) \rightarrow O(n), O(1)$ space.

Definition

Divide & conquer = break a problem into smaller pieces, solve the piece that matters, ignore the rest. **Binary search** applies it to a **sorted** array: look at the middle, then throw away the half that can't contain the target. Real life: finding a word in a **dictionary** – open the middle, go left or right.

find 16 · mid=4

1	3	5	8	10	12	16	20	25
---	---	---	---	----	----	----	----	----

10 < 16 → go right

lo=5 · mid=6

1	3	5	8	10	12	16	20	25
---	---	---	---	----	----	----	----	----

found at index 6 ✓

$n = 1,000,000 \rightarrow$ at most ~20 checks ($2^{20} \approx 1M$)

python

```
def binary_search(arr, target):    # O(log n)
    lo, hi = 0, len(arr) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            lo = mid + 1           # go right
        else:
            hi = mid - 1          # go left
    return -1
```

javascript

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

✎ Best $O(1)$ (first middle) · worst / average $O(\log n)$ (halves every step) · space $O(1)$ iterative, $O(\log n)$ recursive.

Mistakes

- ✎ `while lo < hi` misses the last element → use `lo <= hi`.
- ✎ `lo = mid` / `hi = mid` → infinite loop; always `mid + 1` / `mid - 1`.
- ✎ JS: `(lo + hi) / 2` is a decimal → wrap in `Math.floor`. Python: use `//`.
- ✎ Running it on **unsorted** data gives wrong answers.

Revision

Sorted + search → binary search. lo, hi, mid; equal → return; smaller → lo = mid+1; bigger → hi = mid-1; loop while $lo \leq hi$. $O(\log n)$. Python built-in: `bisect.bisect_left(arr, x)`.

Problem

Write `same(a, b)` → true if every value in `a` has its square in `b`, with the same frequency. Order doesn't matter.

`[1,2,3,2], [9,1,4,4]` → true · `[1,2,3], [1,9]` → false · `[1,2,1], [4,4,1]` → false (frequencies differ)

Way 1 · Brute force – $O(n^2)$

For each `x` in `a`, search for `x2` in `b` (that search is itself a loop) and remove it so it can't be reused.

python

```
def same_brute(a, b):
    # O(n^2)
    if len(a) != len(b):
        return False
    b = list(b)
    # don't mutate input
    for x in a:
        if x * x not in b:
            # .index is O(n)
            return False
        b.remove(x * x)
    return True
```

javascript

```
function sameBrute(a, b) {
    // O(n^2)
    if (a.length !== b.length) return false;
    b = [...b];
    // don't mutate input
    for (const x of a) {
        const idx = b.indexOf(x * x);
        // O(n) scan
        if (idx === -1) return false;
        b.splice(idx, 1);
    }
    return true;
}
```

Way 2 · Frequency counter – $O(n)$

1:1	2:2	3:1
-----	-----	-----

freq of a = [1,2,3,2]

1:1	4:2	9:1
-----	-----	-----

freq of b = [9,1,4,4]

for each key `x` in `freqA` → `freqB[x2]` must equal `freqA[x]`

python

```
def same(a, b): # O(n)
    if len(a) != len(b):
        return False
    fa, fb = {}, {}
    for x in a:
        fa[x] = fa.get(x, 0) + 1
    for y in b:
        fb[y] = fb.get(y, 0) + 1
    for x, cnt in fa.items():
        if fb.get(x * x) != cnt:
            return False
    return True
```

javascript

```
function same(a, b) { // O(n)
    if (a.length !== b.length) return false;
    const fa = {}, fb = {};
    for (const x of a) fa[x] = (fa[x] || 0) + 1;
    for (const y of b) fb[y] = (fb[y] || 0) + 1;
    for (const key in fa) {
        if (fb[key * key] !== fa[key]) return false;
    }
    return true;
}
```

Mistake

Brute force with `indexOf` / `in` looks like one loop but is $O(n^2)$ — the search hides a second loop. And don't mutate the caller's array with `splice` / `remove`; copy it first.

Revision

Compare two collections by value + count → two frequency maps, then compare keys. Three separate loops = $O(3n) = O(n)$. Space $O(n)$.

Definition

Recursion = a function that **calls itself** on a smaller version of the problem until it reaches a case it can answer directly. Real life: **Russian dolls** — open one, there's a smaller one inside, until the tiniest doll.

Every recursive function needs 2 things

1 · Base case

The stopping condition — returns without calling itself. No base case = **infinite recursion**.

2 · Smaller input

Each call must move **closer** to the base case ($n - 1$, `slice(1)`, `half...`).

python

```
def countdown(n):  
    if n <= 0:                # 1. base case  
        print("Done!")  
        return  
    print(n)  
    countdown(n - 1)         # 2. smaller input
```

javascript

```
function countdown(n) {  
    if (n <= 0) {              // 1. base case  
        console.log("Done!");  
        return;  
    }  
    console.log(n);  
    countdown(n - 1);         // 2. smaller input  
}
```

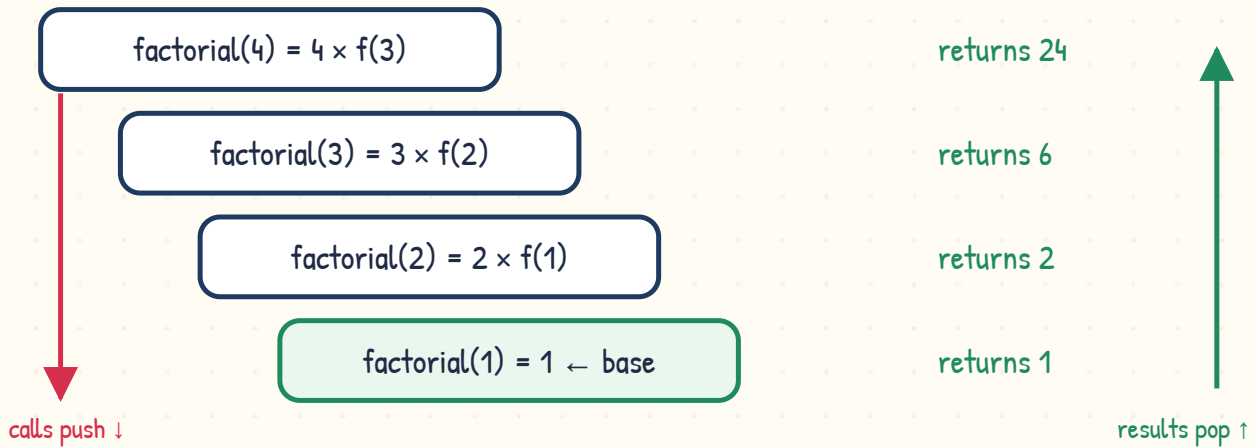
python

```
def factorial(n):              # 0(n) time+stack  
    if n <= 1:  
        return 1  
    return n * factorial(n - 1)  
  
def sum_range(n):             # 1 + 2 + ... + n  
    if n == 1:  
        return 1  
    return n + sum_range(n - 1)
```

javascript

```
function factorial(n) {        // 0(n) time+stack  
    if (n <= 1) return 1;  
    return n * factorial(n - 1);  
}  
  
function sumRange(n) {         // 1 + 2 + ... + n  
    if (n === 1) return 1;  
    return n + sumRange(n - 1);  
}
```

The call stack – factorial(4)



Mistakes

- ✎ No / wrong base case → `RecursionError` (Python, default limit ~1000) / `RangeError: Maximum call stack size exceeded` (JS).
- ✎ Forgetting to return the recursive result – `factorial(n - 1)` without `return` gives `None` / `undefined`.
- ✎ Input that never shrinks (calling with `n` instead of `n - 1`).

Revision

Base case + smaller input + return the result. Each call waits on the stack until the base case answers, then results bubble back up. Stack space = $O(\text{depth})$.

The recipe

Ask: "If someone solved the smaller problem for me, how do I finish?" — $\text{power}(b, e) = b \times \text{power}(b, e-1)$; $\text{product}(\text{arr}) = \text{arr}[0] \times \text{product}(\text{rest})$; $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$.

python

```
def power(base, exp):
    if exp == 0:
        return 1
    return base * power(base, exp - 1)

def product(arr):
    if not arr:
        return 1
    return arr[0] * product(arr[1:])

def fib(n):
    # O(2^n)!
    if n <= 2:
        return 1
    return fib(n - 1) + fib(n - 2)
```

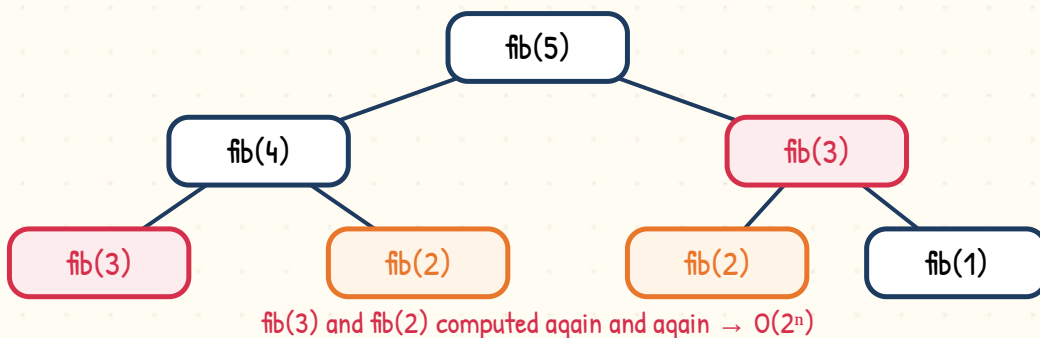
javascript

```
function power(base, exp) {
    if (exp === 0) return 1;
    return base * power(base, exp - 1);
}

function product(arr) {
    if (arr.length === 0) return 1;
    return arr[0] * product(arr.slice(1));
}

function fib(n) {
    // O(2^n)!
    if (n <= 2) return 1;
    return fib(n - 1) + fib(n - 2);
}
```

Why naive Fibonacci is slow



Fix · memoisation — remember answers

python

```
def fib_memo(n, memo=None):
    # O(n)
    memo = {} if memo is None else memo
    if n <= 2:
        return 1
    if n not in memo:
        memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)
    return memo[n]
```

javascript

```
function fibMemo(n, memo = {}) { // O(n)
  if (n <= 2) return 1;
  if (memo[n]) return memo[n];
  memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
  return memo[n];
}
```

Recursion on strings

python

```
def reverse(s):
  if len(s) <= 1:
    return s
  return reverse(s[1:]) + s[0]
```

javascript

```
function reverse(s) {
  if (s.length <= 1) return s;
  return reverse(s.slice(1)) + s[0];
}
```

Python shortcut for memo: `@functools.lru_cache(maxsize=None)` on top of the plain `fib`. Memoisation is the first step into Dynamic Programming.

Mistake

Slicing (`arr[1:]`, `arr.slice(1)`, `s[1:]`) copies the array every call → hidden $O(n)$ per call, $O(n^2)$ total. Fine for learning; in interviews mention passing an index instead.

Revision

Trust the smaller call, finish the last step. Fibonacci tree repeats work → memo with a dict/object turns $O(2^n)$ into $O(n)$.

Definition

A **helper method** pattern = an **outer function** that holds the result (array, counter) and an **inner recursive function** that does the work and writes into it. Useful when you need to **collect** things — because a plain recursive call would reset the result every time.

Problem: collect all **odd** numbers from an array. `[1,2,3,4,5]` → `[1,3,5]`

python

```
def collect_odds(arr):
    result = []                # outer state

    def helper(sub):          # recursive helper
        if not sub:
            return
        if sub[0] % 2 != 0:
            result.append(sub[0])
        helper(sub[1:])

    helper(arr)
    return result
```

javascript

```
function collectOdds(arr) {
    const result = [];        // outer state

    function helper(sub) {    // recursive helper
        if (sub.length === 0) return;
        if (sub[0] % 2 !== 0) result.push(sub[0]);
        helper(sub.slice(1));
    }

    helper(arr);
    return result;
}
```

Same thing with pure recursion

No outer variable: each call **returns** its piece; pieces join on the way back. **Helper:** the result lives outside the recursion, so it isn't reset each call.

python

```
def collect_odds_pure(arr):
    if not arr:
        return []
    first = [arr[0]] if arr[0] % 2 else []
    return first + collect_odds_pure(arr[1:])
```

javascript

```
function collectOddsPure(arr) {
    if (arr.length === 0) return [];
    const first = arr[0] % 2 !== 0 ? [arr[0]] : [];
    return first.concat(collectOddsPure(arr.slice(1)));
}
```

Point	Helper method	Pure recursion
State	Outer variable (closure)	Passed back via return values
Tip	Watch out: <code>result = []</code> inside the recursive fn → reset each call	Use copy methods: <code>+</code> , <code>concat</code> , spread — don't mutate

Revision

Need to collect results? Outer function keeps the list, inner helper recurses and appends. Or pure recursion: return [piece] + recurse(rest).

Definition

Searching = finding the position of a value in a collection. **Linear search** checks items one by one from the start – works on **any** array. Real life: looking for your friend in a queue, person by person.

7	2	9	4	6	1
0	1	2	3	4	5

search 4 → check 7 ×, 2 ×, 9 ×, 4 ✓ → return index 3

python

```
def linear_search(arr, target):    # O(n)
    for i, value in enumerate(arr):
        if value == target:
            return i
    return -1

# built-ins: x in arr, arr.index(x)
```

javascript

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

// built-ins: arr.includes(x), arr.indexOf(x)
```

Point	Linear search	Binary search (Ch 7)
Needs sorted?	No	Yes
Best	$O(1)$ – first element	$O(1)$ – first middle
Worst / avg	$O(n)$	$O(\log n)$
Space	$O(1)$	$O(1)$ iterative
Use when	Small or unsorted data, search once	Large sorted data, many searches
Built-ins	<code>in</code> , <code>.index()</code> , <code>includes</code> , <code>indexOf</code> , <code>find</code>	<code>bisect</code> (Python) · write your own in JS

Mistake

Sorting an array ($O(n \log n)$) **just to** binary-search it once – slower than one linear search ($O(n)$). Sort first only if you'll search **many** times. Also: Python `.index()` throws `ValueError` when missing; JS `indexOf` returns `-1`.

Revision

Linear: any order, $O(n)$. Binary: sorted only, $O(\log n)$. Return index or `-1`. Built-ins are linear too.

Definition

Bubble sort compares neighbours and swaps them if they're in the wrong order. After each full pass, the **largest remaining value** "bubbles up" to the end. Real life: bubbles in a soda — the big ones rise to the top.

One pass on [5, 3, 4, 1, 2]



Basic version (#15)

python

```
def bubble_sort(arr):
    # O(n^2)
    n = len(arr)
    for i in range(n):
        for j in range(n - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

javascript

```
function bubbleSort(arr) {
    // O(n^2)
    const n = arr.length;
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < n - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
            }
        }
    }
    return arr;
}
```

Optimised version (#16)

- ✏️ **Shrink the inner loop:** after pass i , the last i items are already sorted — don't compare them again.
- ✏️ **swapped flag:** if a whole pass made **no swap**, the array is sorted → stop early. Nearly-sorted input becomes $O(n)$.

python

```
def bubble_sort_opt(arr):
    n = len(arr)
    for i in range(n - 1, 0, -1): # shrink end
        swapped = False
        for j in range(i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped: # already sorted
            break # best case O(n)
    return arr
```

javascript

```
function bubbleSortOpt(arr) {
    for (let i = arr.length - 1; i > 0; i--) { // shrink end
        let swapped = false;
        for (let j = 0; j < i; j++) {
            if (arr[j] > arr[j + 1]) {
                [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
                swapped = true;
            }
        }
        if (!swapped) break; // best case O(n)
    }
    return arr;
}
```

Case	Basic	Optimised
Best (already sorted)	$O(n^2)$	$O(n)$ — one pass, no swaps, break
Average / worst	$O(n^2)$	$O(n^2)$
Space · stable?	$O(1)$ in place · stable (equal items keep their order)	

Swap in one line — Python: `a[i], a[j] = a[j], a[i]` · JS: `[a[i], a[j]] = [a[j], a[i]]` (old way: a `temp` variable).

Revision

Compare neighbours, swap, largest bubbles to the end each pass. Optimise: inner loop to i , break when no swap. $O(n^2)$, best $O(n)$, stable, in place.

Definition

Selection sort repeatedly **selects the smallest** value from the unsorted part and swaps it to the front. The sorted part grows from the **left**. Real life: picking the shortest student to stand first, then the next shortest...



green = sorted part · orange = position to fill · purple = smallest found

python

```
def selection_sort(arr):          # O(n^2)
    n = len(arr)
    for i in range(n - 1):
        smallest = i
        for j in range(i + 1, n):
            if arr[j] < arr[smallest]:
                smallest = j
        if smallest != i:          # 1 swap per pass
            arr[i], arr[smallest] = arr[smallest], arr[i]
    return arr
```

javascript

```
function selectionSort(arr) {      // O(n^2)
    const n = arr.length;
    for (let i = 0; i < n - 1; i++) {
        let smallest = i;
        for (let j = i + 1; j < n; j++) {
            if (arr[j] < arr[smallest]) smallest = j;
        }
        if (smallest !== i) {       // 1 swap per pass
            [arr[i], arr[smallest]] = [arr[smallest], arr[i]];
        }
    }
    return arr;
}
```

Point	Selection sort
Time	$O(n^2)$ in all cases — it always scans the whole rest, even if sorted
Swaps	At most $n - 1 \rightarrow$ good when writes/swaps are expensive
Space · stable?	$O(1)$ in place · not stable (a long swap can jump over an equal value)

Mistake

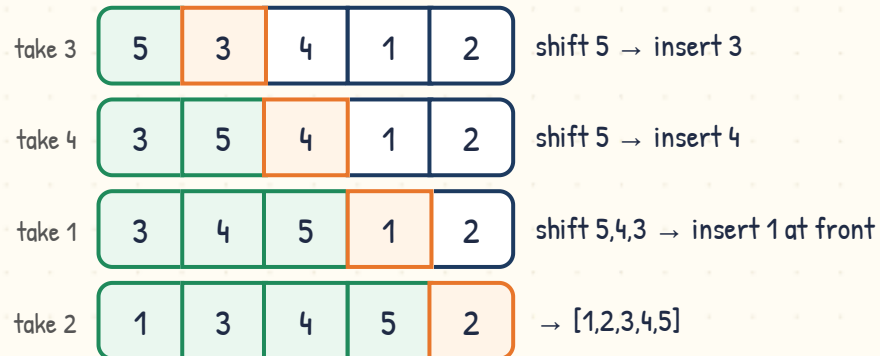
Swapping inside the inner loop every time you see a smaller value — that's not selection sort (and does many more swaps). Track the **index** of the minimum, swap **once** after the inner loop.

Revision

For each position i : find index of min in $i..end$, swap it into i . $O(n^2)$ always, $\leq n-1$ swaps, not stable.

Definition

Insertion sort builds the sorted part one item at a time: take the next item and **insert it into the correct place** in the left part by shifting bigger items right. Real life: **sorting playing cards** in your hand.



python

```
def insertion_sort(arr):          # O(n^2), best O(n)
    for i in range(1, len(arr)):
        current = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > current:
            arr[j + 1] = arr[j]    # shift right
            j -= 1
        arr[j + 1] = current      # drop in the gap
    return arr
```

javascript

```
function insertionSort(arr) {      // O(n^2), best O(n)
    for (let i = 1; i < arr.length; i++) {
        const current = arr[i];
        let j = i - 1;
        while (j >= 0 && arr[j] > current) {
            arr[j + 1] = arr[j];    // shift right
            j--;
        }
        arr[j + 1] = current;      // drop in the gap
    }
    return arr;
}
```

Sorting summary

Algorithm	Best	Average	Worst	Space	Stable	Remember
Bubble (opt.)	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Biggest bubbles to the end
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Fewest swaps
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Best for nearly sorted / small / streaming data
Built-in	Python <code>sorted()</code> / <code>.sort()</code> & modern JS <code>.sort()</code> use TimSort – $O(n \log n)$, stable				Yes	Use this in real code

JS sort gotcha

`[10, 1, 5, 100].sort()` → `[1, 10, 100, 5]` 🤯 – default sort compares as **strings**. For numbers always pass a comparator: `arr.sort((a, b) => a - b)`. Python's `sorted()` compares numbers correctly. Both `.sort()` methods **mutate**; Python `sorted()` / JS `toSorted()` return a new list.

Revision

Insertion: take next item, shift bigger ones right, drop it in the gap. $O(n^2)$, best $O(n)$, stable. All three basic sorts are $O(n^2)$; built-ins are $O(n \log n)$.

Problem

Return the character that appears the most in a string.

"javascript" → "a" (2 times) · "aabbbcc" → "b"

Frequency counter again

j:1	a:2	v:1	s:1	c:1	r:1	i:1	p:1	t:1
-----	-----	-----	-----	-----	-----	-----	-----	-----

pass 1: build counts · pass 2: pick the key with the biggest count

python

```
def max_char(s): # O(n)
    freq = {}
    for ch in s:
        freq[ch] = freq.get(ch, 0) + 1
    best, best_count = '', 0
    for ch, cnt in freq.items():
        if cnt > best_count:
            best, best_count = ch, cnt
    return best

# Counter(s).most_common(1)[0][0]
```

javascript

```
function maxChar(s) { // O(n)
    const freq = {};
    for (const ch of s) freq[ch] = (freq[ch] || 0) + 1;
    let best = "", bestCount = 0;
    for (const ch in freq) {
        if (freq[ch] > bestCount) {
            best = ch;
            bestCount = freq[ch];
        }
    }
    return best;
}
```

- Time $O(n)$. Space $O(k)$ where k = distinct chars (≤ 26 for lowercase letters → effectively $O(1)$).
- Ties: this returns the first char that reached the max (dicts / objects keep insertion order — but JS objects put integer-like keys like "1" first!). Ask the interviewer how to break ties.

Alternative for only lowercase letters: an array of 26 zeros, index `ord(ch) - ord('a')` / `ch.charCodeAt(0) - 97`. Same idea, fixed memory.

Mistake

Sorting the string or using `s.count(ch)` / `split(ch).length` inside a loop → $O(n^2)$. Count once in a map. Also clarify: ignore spaces? case-sensitive?

Revision

Count chars in a dict/object → scan for max. $O(n)$. Python one-liner: `Counter(s).most_common(1)[0][0]`.

Problem

A **palindrome** reads the same forwards and backwards. "madam", "racecar", "121" → true · "abca" → false

Way 1 · Reverse and compare – $O(n)$ time, $O(n)$ space

python

```
s == s[::-1]
```

javascript

```
s === s.split("").reverse().join("")
```

Way 2 · Two pointers – $O(n)$ time, $O(1)$ space



compare the outside pair, move both pointers inward; stop when they meet

python

```
def is_palindrome(s):                #  $O(n)$ ,  $O(1)$ 
    left, right = 0, len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True

# quick: s == s[::-1]
```

javascript

```
function isPalindrome(s) {           //  $O(n)$ ,  $O(1)$ 
    let left = 0, right = s.length - 1;
    while (left < right) {
        if (s[left] !== s[right]) return false;
        left++;
        right--;
    }
    return true;
}

// quick: s === [...s].reverse().join("")
```

Real interview version – ignore case & symbols

python

```
def is_palindrome_clean(s):          # "A man, a plan..."
    s = ''.join(c.lower() for c in s if c.isalnum())
    return s == s[::-1]
```

javascript

```
function isPalindromeClean(s) {      // "A man, a plan..."
    s = s.toLowerCase().replace(/[^a-z0-9]/g, "");
    return s === [...s].reverse().join("");
}
```

Mistake

`str.reverse()` doesn't exist in JS (strings have no reverse) → split → reverse → join, or `[...s]`. In Python `reversed(s)` returns an iterator, not a string — use `s[::-1]` or `''.join(reversed(s))`.

Revision

Reverse & compare (easy, $O(n)$ space) or two pointers from both ends ($O(1)$ space). Clean the string first if spaces/case/punctuation should be ignored.

Longest Substring Without Repeating Characters

Problem (LeetCode #3 · Medium)

Return the length of the longest substring with no repeated characters.

"abcabcbb" → 3 ("abc") · "bbbbbb" → 1 · "pwwkew" → 3 ("wke" — "pwke" is a subsequence, not a substring)

Approaches

Approach	How	Time
Brute force	Check every substring for duplicates	$O(n^2)$
Sliding window + set	Grow right; on duplicate, shrink left one by one	$O(n)$
Sliding window + last-seen map ✓	On duplicate, jump start past its last position	$O(n)$

Variable-size window on "abcabcbb"

end=2 · "abc"	a	b	c	a	b	c	b	b	best = 3
end=3 · 'a' seen at 0	a	b	c	a	b	c	b	b	start → 1
end=6 · 'b' seen at 4	a	b	c	a	b	c	b	b	start → 5
end=7 · 'b' seen at 6	a	b	c	a	b	c	b	b	answer 3

python

```
def longest_unique(s):
    # O(n)
    last_seen = {}          # char -> index
    start = best = 0
    for end, ch in enumerate(s):
        if ch in last_seen and last_seen[ch] >= start:
            start = last_seen[ch] + 1  # jump past dup
        last_seen[ch] = end
        best = max(best, end - start + 1)
    return best
```

```
function longestUnique(s) { // O(n)
  const lastSeen = new Map(); // char -> index
  let start = 0, best = 0;
  for (let end = 0; end < s.length; end++) {
    const ch = s[end];
    if (lastSeen.has(ch) && lastSeen.get(ch) >= start) {
      start = lastSeen.get(ch) + 1; // jump past dup
    }
    lastSeen.set(ch, end);
    best = Math.max(best, end - start + 1);
  }
  return best;
}
```

Mistake – the "abba" trap

Without the check `last_seen[ch] >= start`, the window can move **backwards**: at the last 'a' in "abba", 'a' was seen at 0, which is already outside the window (`start = 2`). Jumping to `0 + 1 = 1` would wrongly give 3. Only jump if the duplicate is **inside** the current window (or use `start = max(start, seen + 1)`).

Revision

Variable sliding window: expand end every step; on a duplicate inside the window, move start to `last_seen + 1`; record `last_seen`; `best = max(best, end - start + 1)`. $O(n)$ time, $O(k)$ space.

Task	Python	JavaScript
Length	<code>len(arr)</code>	<code>arr.length</code>
Loop with index	<code>for i, x in enumerate(arr)</code>	<code>arr.forEach((x, i) => ...)</code> / <code>for (let i = 0; ...)</code>
Loop values	<code>for x in arr</code>	<code>for (const x of arr)</code> (of, not in!)
Count in map	<code>d[x] = d.get(x, 0) + 1</code>	<code>d[x] = (d[x] 0) + 1</code>
Key exists	<code>x in d</code>	<code>x in obj</code> / <code>map.has(x)</code>
Slice / copy	<code>arr[1:3]</code> , <code>arr[:]</code>	<code>arr.slice(1, 3)</code> , <code>[...arr]</code>
Reverse string	<code>s[::-1]</code>	<code>[...s].reverse().join("")</code>
Sort numbers	<code>sorted(arr)</code> / <code>arr.sort()</code>	<code>arr.sort((a, b) => a - b)</code>
Swap	<code>a, b = b, a</code>	<code>[a, b] = [b, a]</code>
Infinity	<code>float('inf')</code>	<code>Infinity</code>
Integer division	<code>7 // 2</code> → 3	<code>Math.floor(7 / 2)</code> → 3
Equality	<code>==</code>	<code>===</code> (always strict)
Unique values	<code>len(set(arr))</code>	<code>new Set(arr).size</code>

Rapid-fire Q&A

Q. What is Big-O?

A. How the number of steps (or memory) grows as input size n grows — worst case, constants dropped.

Q. $O(n)$ vs $O(\log n)$ on 1 million items?

A. ~1,000,000 steps vs ~20 steps.

Q. When do two pointers work?

A. Usually on sorted arrays or when comparing both ends (pairs, palindrome, in-place dedupe).

Q. Frequency counter in one line?

A. Count items in a dict/object, then compare counts — replaces nested loops, $O(n)$.

Q. Fixed vs variable sliding window?

A. Fixed: size k given (max sum of k). Variable: grows/shrinks by a rule (longest substring without repeats).

Q. Why does binary search need sorted data?

A. Discarding half is only safe if everything on one side is known to be smaller/bigger.

Q. Two must-haves of recursion?

A. A base case and a smaller input on every call (and return the result).

Q. Why is naive fib slow and how to fix?

A. Recomputes the same subproblems $\rightarrow O(2^n)$. Memoise $\rightarrow O(n)$.

Q. Helper method recursion?

A. Outer function holds the result, inner recursive helper fills it.

Q. Bubble vs selection vs insertion in one line each?

A. Bubble: swap neighbours, max goes to end. Selection: pick min, swap to front. Insertion: insert next item into sorted left part.

Q. Which simple sort for nearly-sorted data?

A. Insertion sort ($O(n)$ best case). Optimised bubble also $O(n)$ best case.

Q. What does stable sort mean?

A. Equal elements keep their original relative order.

Q. Why is `[10,1,5].sort()` wrong in JS?

A. Default sort compares strings; pass `(a, b) => a - b`.

Q. Substring vs subsequence?

A. Substring = contiguous; subsequence = order kept, gaps allowed.

One-page cheat sheet

patterns

compare counts $\rightarrow$ frequency counter
sorted + pair $\rightarrow$ two pointers
k consecutive $\rightarrow$ fixed window
longest/shortest $\rightarrow$ variable window
sorted + find $\rightarrow$ binary search
repeat subprob $\rightarrow$ recursion + memo

complexities

linear search $O(n)$
binary search $O(\log n)$
bubble/sel/ins $O(n^2)$
built-in sort $O(n \log n)$
dict/set lookup $O(1)$ avg
naive fib $O(2^n) \rightarrow$ memo $O(n)$

Next step

Beyond this playlist: merge sort & quick sort ($O(n \log n)$), linked lists, stacks & queues, hash maps in depth, trees (BST, traversals), heaps, graphs (BFS/DFS), dynamic programming. Practise each pattern on LeetCode: Two Sum, Valid Anagram, Remove Duplicates from Sorted Array, Maximum Average Subarray I, Binary Search, Valid Palindrome.

— End of DSA · keep practising!  —