Big-O Complexity Cheat Sheet: AS/CR Primer

Common Big-O Complexities (AS Table)

Big-O	**Growth Rate**	**Example Pattern**
$O(1)$	Constant	Accessing `arr[i]`, hash map lookup
$O(\log n)$	Logarithmic	Binary search, tree traversal
$O(n)$	Linear	Single loop over array
$O(n \log n)$	Linearithmic	Merge sort, heap sort
$O(n^2)$	Quadratic	Nested loops, naive matrix mult
$O(2^n)$	Exponential	Recursive subset generation
$O(n!)$	Factorial	Permutations, TSP brute-force

AS Tip: Find loops and recursion — they're your signal. **CR Tip:** Use scaling intuition: *"what happens if I double n?"*

How to Analyze Time Complexity (AS Steps)

1. Identify all loops and recursive calls.
2. Estimate how many times each will run.
3. Multiply nested operations.
4. Keep the most dominant term (drop constants).

Example:

```
for (int i = 0; i < n; i++)          // O(n)
  for (int j = 0; j < n; j++)        // O(n)
    doSomething();                   // O(1)
```

Total: $O(n \times n) = O(n^2)$

CR Memory Hooks

- $O(1)$: Pick a card from the top — instant.
- $O(n)$: Flip through every page in a book.
- $O(\log n)$: Keep splitting a phone book in half.
- $O(n \log n)$: Sort a big deck by repeatedly merging halves.
- $O(n^2)$: Everyone shakes hands with everyone else.
- $O(2^n)$: Try every outfit combination from your wardrobe.

Lower Bounds (Proven Limits)

Some operations are provably impossible to improve beyond a certain point.

Task	**Best Possible**	**Why**
Searching in unsorted array	$O(n)$	Must check all elements
Searching in sorted array	$O(\log n)$	Binary search is optimal
Comparison sorting	$O(n \log n)$	Proven lower bound
Hash lookup (avg case)	$O(1)$	But worst-case $O(n)$
Matrix multiplication	$O(n^{2.3729})$	No $O(n \log n)$ known
Subset sum / TSP / SAT	$O(2^n)$	NP-complete; exponential only

CR Visualization: Scaling Impact

Assume an algorithm takes 1ms at $n = 100$.

Complexity	**Time at** $n = 10,000$
$O(1)$	1ms
$O(\log n)$	2ms
$O(n)$	100x $\rightarrow$ 100ms
$O(n \log n)$	132ms
$O(n^2)$	10,000x $\rightarrow$ 10,000ms = 10s
$O(2^n)$	*unimaginable*

Tips for Choosing Algorithms

- Use $O(1)$ or $O(\log n)$ if data is indexed or sorted.
- Aim for $O(n)$ if you must read all items.
- Accept $O(n \log n)$ for sorting or divide-and-conquer.
- Avoid $O(n^2)$+ unless $n \leq 1000$.
- Avoid $O(2^n)$+ unless $n \leq 20$ or you're pruning search.

Final Rule of Thumb

If your algorithm is:

- $O(1)$ or $O(\log n)$: **scales beautifully**
- $O(n)$ or $O(n \log n)$: **usually safe**
- $O(n^2)$ or worse: **danger zone beyond 10,000 items**

Part 1 — Practice Problems

Estimate the **worst-case time complexity** in Big-O for each function below. Assume input size is n.

1. **Single loop**

```
void func1(int n) {
    for (int i = 0; i < n; ++i) {
        std::cout << i;
    }
}
```

2. **Nested loop**

```
void func2(int n) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            std::cout << i << j;
}
```

3. **Logarithmic growth**

```
void func3(int n) {
    while (n > 1) {
        n = n / 2;
    }
}
```

4. **Recursive Fibonacci (naive)**

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

5. **Merge sort-like pattern**

```
void mergeSort(int arr[], int n) {
    if (n <= 1) return;
    int mid = n / 2;
    mergeSort(arr, mid);
    mergeSort(arr + mid, n - mid);
    // merge step (assume linear time)
}
```

6. **Two separate loops**

```
void func6(int n) {
    for (int i = 0; i < n; ++i)
        std::cout << i;
    for (int j = 0; j < n; ++j)
        std::cout << j;
}
```

Part 2 — Answer Key and Explanations

1. **Single loop** — $\boxed{O(n)}$

   The loop runs exactly n times. Each step is $O(1)$.

   —

   2. Nested loop — $\boxed{O(n^2)}$

   Inner loop runs n times per outer loop. Total steps: $n \times n$.

   —

   3. Logarithmic growth — $\boxed{O(\log n)}$

   Each iteration halves n. Number of steps is log base 2 of n.

   —

   4. Recursive Fibonacci — $\boxed{O(2^n)}$

   Two recursive calls per level, forms an exponential tree.

   —

   5. Merge sort — $\boxed{O(n \log n)}$

   Divide: $\log n$ levels; Merge: $O(n)$ per level. Total: $n \log n$.

   —

   6. Two separate loops — $\boxed{O(n)}$

   First loop is $O(n)$, second is $O(n)$. Add them $\rightarrow$ still $O(n)$.