PROBLEM SOLVING

Simple examples for each major time complexity.

Time Complexity	Examples	Explanation
O(1) (Constant Time)	- Accessing an array element arr[5] - Swapping two variables - Checking if a number is even (x % 2 == 0)	Time does not depend on input size
O(log n) (Logarithmic Time)	- Binary search in a sorted array - Finding the height of a balanced BST - Halving a number until 1 (e.g. while n > 1: n //= 2)	Input is reduced by half each step
O(n) (Linear Time)	- Finding max in an array - Linear search - Printing all elements of an array	Time grows linearly with input size
O(n log n) (Linearithmic Time)	- Merge Sort - Heap Sort - Efficient sorting algorithms	Combination of linear and logarithmic behavior
$O(n^2)$ (Quadratic Time)	- Bubble Sort - Insertion Sort - Nested loops over array: for i in n: for j in n:	Time grows exponentially with input
$O(n^3)$ (Cubic Time)	- Matrix multiplication (naive) - 3 nested loops: for i in n: for j in n: for k in n:	Time increases very fast with input size
$O(2^n)$ (Exponential Time)	- Solving Tower of Hanoi - Recursive Fibonacci - Generating all subsets	Time doubles with each added input
O(n!) (Factorial Time)	- Solving Traveling Salesman Problem by brute force - Generating all permutations - Recursive solutions without pruning	Extremely slow growth, infeasible for n > 10

Example Code Snippets for Clarity

O(1):

```
int getFirst(int arr[]) {
    return arr[0];
}
```

O(log n):

1)
```
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n-1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key) return mid;
        else if (arr[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

2)
```
for(int i = 1; i < n; i *= 2) {
    printf("%d", i);
}
```

O(n):

1)
```
for(int i = 0; i < n; i++) {
    printf("%d", i);
}
```

2)
```
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);}
```

$O(n^2)$:

1)
```
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        printf("%d, %d", i, j);
    }
}
```

2)
```
void printPairs(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("(%d,%d) ", arr[i], arr[j]);
        }}}
```

$O(n \log n)$:

```
for(int i = 0; i < n; i++) {
    for(int j = 1; j < n; j *= 2) {
        printf("%d, %d", i, j);
    }
}
```

$O(2^n)$

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

Use Master Theorem for recursive functions like:

$T(n) = 2T(n/2) + O(n) \rightarrow O(n \log n)$

$T(n) = T(n/2) + O(1) \rightarrow O(\log n)$

Time Complexity Identification Tricks (with C examples)

Pattern	Time Complexity	Example	Explanation
Single loop over n	O(n)	for(i = 0; i < n; i++)	Runs n times
Two nested loops	$O(n^2)$	for(i = 0; i < n; i++) for(j = 0; j < n; j++)	n * n iterations
Three nested loops	$O(n^3)$	for(i=0;i<n;i++) for(j=0;j<n;j++) for(k=0;k<n;k++)	n * n * n iterations
Loop dividing n by 2 each time	O(log n)	for(i = n; i > 1; i /= 2)	Halves each step
Loop multiplying i by 2	O(log n)	for(i = 1; i < n; i *= 2)	Doubles each step
Loop inside loop with decreasing range	O(n log n)	for(i = 0; i < n; i++) for(j = n; j > 1; j /= 2)	n * log n
Recursive Fibonacci	$O(2^n)$	fib(n) = fib(n-1) + fib(n-2)	2 calls per level
Generating permutations	O(n!)	Backtracking or recursion over all orderings	Factorial growth

Tricks for Quick Analysis

- ❖ Count how many times the loop runs (visually)

  E.g. for(i = 1; i < n; i *= 2)
- ❖ Doubles → Logarithmic → O(log n)
- ❖ Nested loops = Multiply inner by outer

```
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
```

→ Outer loop n, inner loop n = $O(n^2)$

- ❖ Check if recursive calls grow exponentially or divide

recur(n) = recur(n-1) + recur(n-2) → $O(2^n)$

recur(n) = recur(n/2) → O(log n)

recur(n) = 2 * recur(n/2) → O(n)

- ❖ Sorting usually = O(n log n) (e.g. Merge Sort, Quick Sort average case)
- ❖ Use Log Table:

i = 1; while(i < n) { i *= 2; } // $\log_2 n$ times

i = n; while(i > 0) { i /= 2; } // $\log_2 n$ times