
10101
0110

Contents

PROBLEM TYPE	COMMON APPROACH	TYPICAL TIME COMPLEXITY	SPACE COMPLEXITY
STRING/ARRAY SEARCH	*Two Pointers/Sliding Window*	O(n)	O(1) to O(n)
MATRIX TRAVERSAL	*BFS/DFS*	O(n · m)	O(n · m)
BINARY SEARCH	*Divide & Conquer*	O(log(n))	O(1)
TREE TRAVERSAL	*BFS/DFS*	O(n)	O(h) where h is height
GRAPH SHORTEST PATH	*BFS/Dijkstra's*	O(V + E) or O((V + E) · log(V))	O(V)
SUBSTRING PROBLEMS	*Sliding Window/DP*	O(n) to $O(n^2)$	O(1) to O(n)
PERMUTATIONS/COMBINATIONS	*Backtracking*	O(n!) or $O(2^n)$	O(n)
DYNAMIC PROGRAMMING	*Memoization & Tabulation*	Problem dependent	Problem dependent
HEAP PROBLEMS	*Priority Queue*	O(n · log(n))	O(n)
TOPOLOGICAL SORT	*BFS/DFS*	O(V + E)	O(V)
UNION FIND / DISJOINT SET	*Path compression & union by rank*	O(α(n))	O(n)
BIT MANIPULATION	*Bitwise operations*	O(1) to O(log(n))	O(1)
GREEDY ALGORITHMS	*Local optimization at each step*	O(n · log(n))	O(1) to O(n)
TRIE PROBLEMS	*Prefix tree construction/traversal*	O(m)	O(n · m)
INTERVAL PROBLEMS	*Sort + Line sweep*	O(n · log(n))	O(n)
MONOTONIC STACK/QUEUE	*Maintaining increasing/decreasing sequence*	O(n)	O(n)

KEYWORD/PATTERN	ALGORITHM	KEYWORD/PATTERN	ALGORITHM
"TOP K"	Heap	"HOW MANY WAYS..."	DFS, DP
"SUBSTRING"	Sliding Window	"PALINDROME"	Two Pointers, DFS, DP
"TREE"	BFS (shortest, level-order), DFS (otherwise)	"PARENTHESES"	Stack
"SUBARRAY"	Sliding Window, Prefix Sum, HashMap	"MAX SUBARRAY"	Greedy
"X SUM"	Two Pointers	"MAX/LONGEST SEQUENCE"	DFS, DP, Monotonic Deque
"MINIMUM/SHORTEST"	DFS, DP, BFS	"PARTITION/SPLIT ... ARRAY/STRING"	DFS
"SUBSEQUENCE"	DFS, DP, Sliding Window	"MATRIX"	BFS, DFS, DP
"JUMP"	Greedy/DP	"GAME"	Dynamic Programming
"CONNECTED COMPONENT", "CUT/REMOVE", "REGIONS/GROUPS/CONNECTIONS"	Union Find	"TRANSITIVE RELATIONSHIP"	BFS, Union Find
"INTERVAL"	Greedy (Sort by start/end time)		

DATA STRUCTURE	TIME COMPLEXITY								SPACE COMPLEXITY
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
ARRAY	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
STACK	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
QUEUE	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
SINGLY-LINKED LIST	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
DOUBLY-LINKED LIST	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
SKIP LIST	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n · log(n))
HASH TABLE	O(1)	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)	O(n)
BINARY SEARCH TREE	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
CARTESIAN TREE	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-TREE	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
RED-BLACK TREE	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
SPLAY TREE	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)

ALGORITHM	TIME COMPLEXITY			SPACE COMPLEXITY	
	Best	Average	Worst	Worst	Is in place
SELECTION SORT	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Yes
BUBBLE SORT	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
INSERTION SORT	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
HEAP SORT	O(n · log(n))	O(n · log(n))	O(n · log(n))	O(1)	Yes
QUICK SORT	O(n · log(n))	O(n · log(n))	$O(n^2)$	O(log(n))	Yes
MERGE SORT	O(n · log(n))	O(n · log(n))	O(n · log(n))	O(n)	No
BUCKET SORT	O(n + k)	O(n + k)	$O(n^2)$	O(n)	No
RADIX SORT	O(n · k)	O(n · k)	O(n · k)	O(n + k)	No
COUNT SORT	O(n + k)	O(n + k)	O(n + k)	O(k)	No
SHELL SORT	O(n · log(n))	$O(n \cdot (\log(n))^2)$	$O(n \cdot (\log(n))^2)$	O(1)	Yes
TIM SORT	O(n)	O(n · log(n))	O(n · log(n))	O(n)	No
TREE SORT	O(n · log(n))	O(n · log(n))	$O(n^2)$	O(n)	No
CUBE SORT	O(n)	O(n · log(n))	O(n · log(n))	O(n)	No

Algorithm Complexity Cheatsheet

Basic Operations

Constant-time operations (O(1)) include arithmetic operations, assignments, comparisons, and accessing array elements by index. Modulo and basic math operations are usually O(1) for standard integer sizes. Bitwise operations are typically O(1), but may be O(b) for b bits in some contexts.

Loops and Iterations

Simple loops have a complexity of O(n) where n is the number of iterations. For nested loops, multiply the complexities (e.g., two nested loops = $O(n^2)$). When dealing with loops with variable bounds, consider the sum of iterations (e.g., triangular loops). For loops with early termination, consider both average and worst case scenarios.

Loops with variable increments behave differently:

- `i++`: O(n)
- `i += 2` or `i *= 2`: Still O(n) (just a constant factor difference)
- `i = i*2` or `i = i^2`: Logarithmic or better, analyze carefully

Recursive Functions

Simple recursion has a complexity of $O(branches^{depth})$. For recursion with overlapping subproblems, analyze the recurrence relation. Tail recursion is often equivalent to an iterative solution. Recursive functions with decreasing inputs (like n/2, n/3) often have O(log(n)) complexity.

Geometric series recursion (sum like n + n/2 + n/4 + ...) is O(n). Multiple recursion, like Fibonacci, can be exponential without memoization. For divide-and-conquer recurrences, consider applying the Master theorem.

Data Structures

Hash tables' performance depends heavily on the hash function and collision resolution strategy. Skip lists are probabilistic data structures with expected O(log(n)) performance. Tree structures such as balanced trees (AVL, Red-Black) guarantee O(log(n)) operations. KD trees' performance degrades in high dimensions due to the "curse of dimensionality."

Sorting Algorithms

Quicksort is often fastest in practice due to good cache locality and low constant factors. Mergesort provides stable sort with guaranteed $O(n \cdot log(n))$ performance. Timsort, a hybrid algorithm used in Python and Java, adapts to patterns in the data.

Non-comparison sorts (Bucket, Radix, Counting) can achieve linear time but with constraints on input. Note that k represents the range of input values or number of digits/bits.

Special Cases

Numbers and Digits

Processing digits of a number: O(log(n)) because an n-value number has $\log_{10}(n)$ digits

Computing a^b: O(b) with simple multiplication, O(log b) with binary exponentiation

Division by repeated addition: O(a/b)

GCD (Euclidean algorithm): O(log (min(a,b)))

Trees and Graphs

Balanced BST operations have O(log(n)) complexity, while unbalanced trees can degrade to O(n) in the worst case. Full tree traversal requires O(n) time.

For graphs, BFS/DFS has O(V + E) complexity where V is vertices and E is edges. Dijkstra's algorithm runs in $O((V + E) \log(V))$ with a binary heap implementation. Bellman-Ford has $O(V \cdot E)$ complexity, while Floyd-Warshall requires $O(V^3)$.

Minimum spanning tree algorithms typically run in $O(E \cdot \log(V))$ with a binary heap. Union-find operations have $O(\alpha(n))$ complexity per operation (where α is the inverse Ackermann function, which is nearly constant).

Dynamic Programming

1D dynamic programming usually has O(n) or $O(n^2)$ complexity. 2D dynamic programming often requires $O(n \cdot m)$ or $O(n^2)$ time. When analyzing DP problems, identify the number of states and transitions through state space analysis.

Amortized Analysis

Dynamic array resizing provides O(1) amortized per insertion. Hash table resizing also achieves O(1) amortized per operation. For more complex data structures, consider using the potential method for analysis.

Space Complexity

The recursive call stack usually requires O(depth of recursion) space. Memoization tables typically need O(number of unique subproblems) space. In-place algorithms use O(1) extra space. When using auxiliary data structures, consider both input size and algorithm needs.

Common Gotchas

Input-dependent complexity: When runtime depends on values (like O(a/b)), not just size

Nested operations: Array copying in a loop can lead to quadratic complexity

String operations: String concatenation in a loop is often quadratic

Recursive calls with shrinking parameters: Often logarithmic or linear (geometric series)

Hidden Loops in Python

- String operations: `+` concatenation in loops, `str.join()`, `str.format()`, `f-strings`
- List operations: `in` operator (O(n) for lists), `+` for list concatenation, list comprehensions
- Built-in functions:
  - `sorted()` (O(n · log(n)))
  - `min()`/`max()`/`sum()` (all O(n))
- Object methods: `__str__`, `__repr__`, `__eq__ (==)`, `__lt__ (<)`
- Library operations: `pandas.apply()`, `copy.deepcopy()`, exhausting generators multiple times
- Always consider the complexity of queries in database-related problems. Know the complexity of library methods you're using.

Algorithmic Patterns

The two-pointer technique often achieves O(n) complexity. Sliding window algorithms usually run in O(n) time. Divide and conquer approaches often result in O(n · log(n)) complexity but depend on the specific problem.

Greedy algorithms typically require O(n · log(n)) time due to sorting. Backtracking is often exponential $O(b^d)$ where b is the branching factor and d is the depth. Prefix sums/difference arrays use O(n) preprocessing to enable O(1) range queries.

Analysis Tips

1. Identify the input variables that affect runtime
2. Analyze loops and find how many iterations they perform
3. For nested operations, multiply rather than add complexities
4. With recursion, identify the recurrence relation and solve it
5. Watch for logarithmic patterns (dividing in half, etc.)
6. Don't forget about the cost of operations inside loops
7. Consider both average and worst case scenarios
8. Watch for amortized complexity in data structure operations
9. Analyze space complexity separately from time complexity
10. For large inputs, constant factors matter less than asymptotic growth

Mathematics for technical interviews

Number Systems and Bases

Base 10 (Decimal)

The decimal system uses 10 unique digits (0-9). Each position represents a power of 10.
For example, the number 352 can be broken down as $(3 \cdot 10^2) + (5 \cdot 10^1) + (2 \cdot 10^0)$.

Base 2 (Binary)

Binary uses only 2 unique digits (0-1). Each position represents a power of 2.
For example, 1011 equals $(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 8 + 0 + 2 + 1 = 11$.
Binary is fundamental in computing because computers operate using binary due to the electronic logic of circuits.

Interview Applications

- Bit manipulation questions
- Understanding memory allocation
- Handling overflow conditions

Logarithms

Concept

A logarithm answers the question: to what power must we raise a base to get a number? If $y = b^x$, then $\log_b(y) = x$.

Binary Logarithm (Base 2)

This represents how many times we need to multiply 2 to reach a number.
For instance, $\log_2(8) = 3$ because $2^3 = 8$, and $\log_2(16) = 4$ because $2^4 = 16$.

An alternative way to think about it is how many times we can divide a number by 2 until we reach 1.
For example, $8 \div 2 \div 2 \div 2 = 1$, so $\log_2(8) = 3$.

Interview Applications

- Time complexity analysis: Many efficient algorithms have logarithmic complexity
- Binary search: O(log(n))
- Divide and conquer algorithms
- Data structures: Height of balanced trees

Permutations and Factorial

Definitions

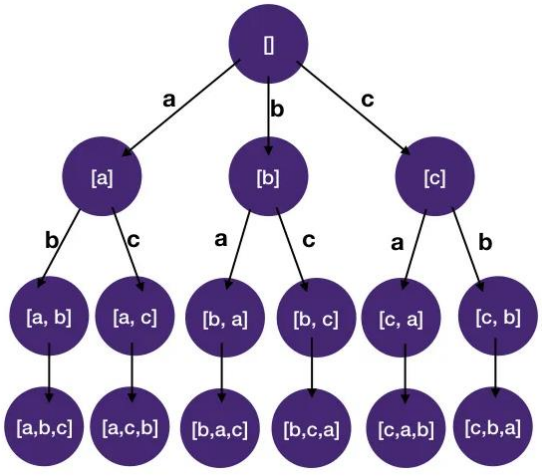
A set is a collection of distinct elements where order doesn't matter. A permutation is a specific arrangement or ordering of elements where order matters.

Counting Permutations

The formula for counting permutations is $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$.
For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.
For n elements, the number of ways to arrange them is n!.

Interview Applications

- Backtracking problems: Generate all permutations
- Counting problems: How many possible ways to arrange/order items



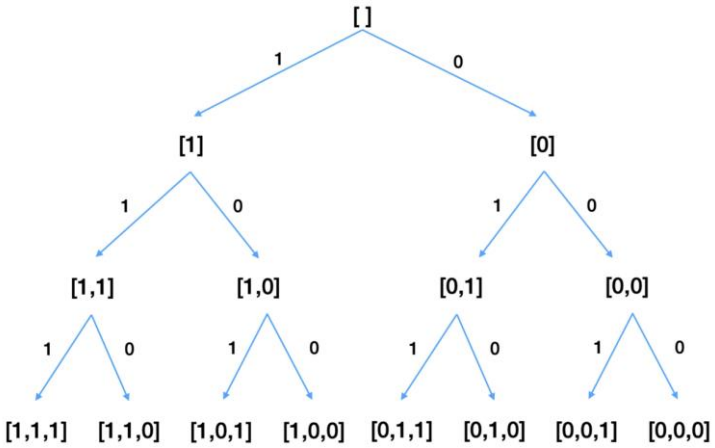
Subsets

Definition

A subset is a set containing only elements that are also in another set. For example, {1, 3, 9} is a subset of {1, 2, 3, 5, 6, 7, 9}.

Counting Subsets

A set with n elements has 2^n subsets.
This is because for each element, we make a binary choice (include/exclude). With n elements, we have 2^n possible combinations.



Interview Applications

- Combinatorial problems: Generate all possible subsets
- Dynamic Programming: Subset sum, knapsack problems
- Bit manipulation: Using bits to represent inclusion/exclusion

Arithmetic Sequences

Definition

An arithmetic sequence is a sequence where the difference between consecutive terms is constant.
Examples include 1, 2, 3, 4, 5 (difference = 1) and 1, 3, 5, 7, 9 (difference = 2).

Sum Formula

The formula for the sum is (first_element + last_element) · number_of_elements ÷ 2.
For example, sum([1,2,3,4,5]) = (1 + 5) · 5 ÷ 2 = 15.

Interview Applications

- Nested loop analysis: Calculating runtime complexity

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

This runs 1+2+...+n times = $O(n^2)$

Geometric Sequences

Definition

A geometric sequence is a sequence where the ratio between consecutive terms is constant.
Examples include 1, 2, 4, 8, 16 (ratio = 2) and 1, 3, 9, 27, 81 (ratio = 3).

Sum Formula

The formula is first_element · (1 - $ratio^{number\_of\_elements}$) ÷ (1 - ratio).

Interview Applications

- Tree problems: Counting nodes in perfect binary trees
- Algorithmic analysis: Some divide and conquer algorithms

Modular Arithmetic

Definition

Modular arithmetic is a system where numbers "wrap around" after reaching a certain value (modulus).
The notation a % b = remainder, when a is divided by b. For example, 15 % 12 = 3.

Properties

Calculation:

If x < y, then x % y = x

otherwise, subtract y from x until x < y

Distributivity:

(a + b) % c = ((a % c) + (b % c)) % c

Interview Applications

- Primality testing: Testing if a number is prime

```
for (i = 2; i <= sqrt(n); i++) {
       if (n % i == 0) return false;// Not prime
}

return true; // Is prime
```

- Cyclic problems: Problems involving repeating patterns
- Hash functions: Implementation of some hash functions

Interview Tip: Connect Mathematical Concepts to Problems

When approaching algorithm problems, identify which mathematical concepts apply:

- **Logarithmic complexity?** Think binary search, divide and conquer
- **Factorial patterns?** Think permutations and backtracking
- **Power of 2 patterns?** Think subsets or bit manipulation
- **Arithmetic patterns?** Analyze nested loops
- **Modular arithmetic?** Consider problems with cyclic patterns or divisibility

Remember that understanding the mathematical foundations helps you recognize problem patterns rather than memorizing specific solutions.

Big O

Time Complexity

> *Airplane analogy - For a big enough file, it's more effective to fly across the country to transfer the hard drive, rather than send it electronically. Electronic transfer O(n), where n is the size of the file (linear time). Airplane transfer O(1), with respect to the size of the file. No matter the file size, the time it takes to transfer it is the same (constant time).*

There can also be multiple variables in the runtime.
If you were painting a fence W x H meters, it could be described as $O(w \cdot h)$.
If you needed P layers of paint, you could say the time is $O(w \cdot h \cdot p)$.

Big O (oh), Big θ (theta), Big Ω (omega)

Big O → upper bound on the time. An algorithm that prints all values in an array could be described as O(n), but it could also be described as $O(n^2)$, $O(n^3)$, or $O(2^n)$. The algorithm is at least as fast as each of these, similar to a less-than-or-equal-to relationship.

Big Ω → lower bound on the time. Printing values in an array is Ω(n), as well as Ω(log(n)), and Ω(1). You know it will never be faster than those runtimes.

Big θ → both lower and upper bounds. An algorithm is θ(n), if it is both O(n) and Ω(n). θ gives a tight bound description on runtime.

Best, worst, and expected case

Best case → If all elements are equal, then quick sort, for example, will just traverse through the array once. This is O(n).

Worst case → If the pivot is repeatedly the biggest element in the array when quick sorting, for example, the recursion doesn't divide the array in half and recurse on each half. It just shrinks the subarray by one element. This is $O(n^2)$.

Expected case → Sometimes the pivot will be very low or very high, but it doesn't happen repeatedly. We can expect a runtime of $O(n \cdot \log(n))$.

For most algorithms, the worst and expected cases are the time. Sometimes they aren't, in which case we need to describe both runtimes.

Space Complexity

Space complexity refers to the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the input. This includes the memory space used by its inputs, called input space, and any other (auxiliary) memory it uses during execution, which is called auxiliary space.

Space complexity is a parallel concept to time complexity - if we need to create an array of size n, this will require O(n) space. If we need a 2D array of size n x n, this will require $O(n^2)$ space.

> *The call stack is specifically a memory structure that tracks function calls, storing local variables, parameters, return addresses, and execution context for each active function in a program's execution flow.*

Recursive sum function

Memory stack space in recursive calls counts, as well.

A recursive sum function creates a chain of calls, where each call remains active on the memory stack while waiting for its nested call to complete. This creates a memory stack that's n levels deep, requiring O(n) space because all calls exist simultaneously in memory.

```
int sum(int n) {
    if (n <= 0) {
       return 0;
    }
    return n + sum(n-1);
}
```

```
sum(4)
  -> sum(3)
    -> sum(2)
      -> sum(1)
        -> sum(0)
```

Iterative function with helper

Just because you have n calls, doesn't mean it takes O(n) space.

The function makes O(n) calls to **`pairSum()`**, but each call to **`pairSum()`** completes and returns before the next call begins. Only one **`pairSum()`** call exists on the memory stack at any given time, so the function uses O(1) space regardless of input size.

```
int pairSumSequence(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += pairSum(i, i + 1);
    }
    return sum;
}

int pairSum(int a, int b) {
    return a + b;
}
```

Space complexity in recursive functions like in example 1 is O(n) because each recursive call remains on the memory stack until the entire chain completes, creating n simultaneous memory frames. In example 2 it's O(1) space complexity because each **`pairSum()`** call completes and is removed from memory before the next one begins, maintaining a constant memory stack depth regardless of input size.

Drop the constants & non-dominant terms

We drop constants and non-dominant terms in Big O notation because Big O describes the rate of growth with respect to input size, not absolute performance.

Though the second example uses two loops (technically 2n operations), both are O(n) since we drop constants.

```
// Min and Max 1
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x : array) {
       if (x < min) min = x;
       if (x > max) max = x;
}
```

```
// Min and Max 2
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x : array) {
       if (x < min) min = x;
}
for (int x : array) {
       if (x > max) max = x;
}
```

With an expression such as $O(n^2 + n)$, the second n isn't exactly a constant, but it's not especially important because it's slower than the first term. We keep only the fastest-growing term since it dominates the runtime as input grows in cases of addition:

- $O(n^2 + n)$ becomes $O(n^2)$
- $O(n + \log(n))$ becomes $O(n)$
- $O(5 \cdot 2^n + 1000 \cdot n^{100})$ becomes $O(2^n)$

The expression $O(B^2 + A)$ cannot be simplified without knowing the relationship between A and B.

Multi-part algorithms: Add vs. Multiply

When do you multiply runtimes, and when do you add them?

```
# Adding runtimes: O(A + B)
for (int a : arrA) {
       print(a);
}

for (int b : arrB) {
       print(b);
}
```

```
# Multiplying runtimes: O(A * B)
for (int a : arrA) {
       for (int b : arrB) {
              print (a + ", " + b);
       }
}
```

We do A chunks of work, then B chunks of work → O(A + B)

If the algorithm is in the form *"do this, then when you're done, do that"* then you add the runtimes.

We do B chunks of work for each element in A → O(A · B)

If the algorithm is in the form *"do this for each time you do that"*, then you multiply the runtimes.

Amortized time

Amortized time allows us to describe that, yes, this worst case happens every once in a while, but once it happens, it won't happen again for so long that the cost is "amortized" across all operations.

With dynamically resizing arrays, you have the benefit of an array while offering flexibility in its size. You won't run out of space since its capacity will grow as you insert elements. When the array hits capacity, it will create a new array with double the capacity and copy all the elements over to the new array (in case of **`ArrayList`** in Java or **`list`** in Python).

If the array contains n elements, then inserting a new element when at capacity will take O(n) time. You will have to create a new array of size 2n and then copy n elements over. This specific insertion will take O(n) time. However, the vast majority of insertions will be O(1) time. As we insert elements, we double the capacity when the size of the array is a power of 2. So after X elements, we double the capacity at array sizes 1, 2, 4, 8, 16, ..., X. That doubling takes, respectively, 1, 2, 4, 8, 16, 32, 64, ..., X copies.

If you read this sum 1 + 2 + 4 + 8 + 16 + ... + X left to right, it starts with 1 and doubles until it gets to X.

If you read right to left, it starts with X and halves until it gets to 1. The sum of X + X/2 + X/4 + X/8 + ... + 1 is approximately 2X.

Therefore, X insertions take O(2X) time, which simplifies to O(X) time. The amortized time for each insertion is O(1). If you perform X operations and they collectively take O(X) time, then on average, each individual operation takes O(1) time.

To find the amortized time per operation, we divide the total cost by the number of operations:

O(X) total time / X operations = O(1) per operation.

Log N runtimes

Where the number of elements in the problem space gets halved each time, that will likely be a O(log(n)) problem.

Binary search in a sorted array is the best example - with each comparison, we go either left or right. Half the nodes are on each side, so we cut the problem space in half each time. We are looking for an example x in an n-element sorted array. We first compare x to the midpoint of the array.

- If **`x == middle`**, then we return. If **`x < middle`**, then we search the left side of the array.
- If **`x > middle`**, then we search on the right side of the array.

We start off with an n element array to search. Then, after a single step, we're down to n/2 elements. One more step, we're down to n/4 elements. We stop when we either find the value or we're down to just one element.

```
search 9 within {1, 5, 8, 9, 11, 13, 15, 19, 21}
   compare 9 to 11 -> smaller
   search 9 within {1, 5, 8, 9, 11}
       compare 9 to 8 -> bigger
       search 9 within {9, 11}
          compare 9 to 9
              return
```

The total runtime is then a matter of how many steps (dividing n by 2 each time) we take until n becomes 1.

Alternatively, how many multiplications by 2 does it take to reach n? In the expression 2^k = n, the k is what log expresses.

$2^4 = 16 \rightarrow \log_2 16 = 4$

$\log_2 n = k \rightarrow 2^k = n$

Recursive runtimes

When analyzing recursive functions with multiple recursive calls, look at the pattern of calls rather than making assumptions.

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

People often incorrectly assume $O(n^2)$ runtime by seeing two recursive calls.

For recursive functions with multiple calls (often, but not always) be: $O(branches^{depth})$

In this example, it's $O(2^n)$.

1. Trace the recursion tree
    a. Each call branches into multiple subcalls
    b. b. f(4) calls f(3) twice, each f(3) calls f(2) twice, etc.
2. Count nodes by level
3. Sum all nodes
    a. Total nodes
        $2^0 + 2^1 + 2^2 + ... + 2^n$
    b. b. This sum equals
        2^(n+1) - 1

Level	# Nodes	Also expressed as...	Or...
0	1		2^0
1	2	2 * previous level = 2	2^1
2	4	2 * previous level = 2·2^1	2^2
3	8	2 * previous level = 2·2^2	2^3

Common time complexities & their applications

EFFICIENT COMPLEXITIES	LESS EFFICIENT COMPLEXITIES
O(1)	O(n · log(n))
O(log(n))	$O(n^2)$
O(n)	$O(2^n)$
O(k · log(n))	O(n!)

INPUT SIZE (N)	O(1)	O(log(n))	O(n)	O(k · log(n))	O(n · log(n))	$O(n^2)$	$O(2^n)$	O(n!)
1	1	0.00	1	0.00	0.00	1	2	1
5	1	2.32	5	23.22	11.61	25	32	120
10	1	3.32	10	33.22	33.22	100	1,024	3.63e+6
20	1	4.32	20	43.22	86.44	400	1.05e+6	2.43e+18
50	1	5.64	50	56.44	282.19	2,500	Too large	Too large
100	1	6.64	100	66.44	664.39	10,000	Too large	Too large
1000	1	9.97	1000	99.66	9965.78	1,000,000	Too large	Too large

Assumptions: k=10 for O(k · log(n)), log = $\log_2$

O(1)

Constant time complexity.

- Hashmap lookup, array access, stack push/pop...
- Mathematical formulas (e.g., sum of first n numbers).
- **Typical Use Case:** handles input sizes up to any magnitude, even n > 10^9

Example, executes a constant number of operations:

```
# All operations are O(1),
# regardless of input size.
result = (n * (n + 1)) // 2
# Sum of 1 to n via formula
```

O(log(n))

Logarithmic time complexity.
Grows very slowly ($\log(10^6) \approx 20$).

- Binary search, balanced BST (AVL, Red-Black tree) operations...
- Divide-and-conquer algorithms where work halves each step.
- Processing the digits of a number.
- Typical Use Case: n > 10^8

```
n = 10e6
# Halving N each iteration → O(log(n))
while n > 0:
    n //= 2
```

O(n)

Linear time complexity.
Typically looping through a linear data structure a constant number of times.

- Single loop over array/list, two pointers, BFS/DFS, stack/queue...
- Typical use case: n ≤ 10^6

```
sum = 0
for i in range(n):       # O(n)
    sum += i
for j in range(2*n):     # Still O(n)
    sum += j
```

O(n · log(n))

Linearithmic time.

- Comparison-based sorting (merge sort, heapsort...)
  - The default sorting algorithm's expected runtime in all mainstream languages is O(n · log(n)).
- Divide-and-conquer with linear merge steps
  - Divide is normally log(n)
  - If merge is O(n) then the overall runtime is O(n · log(n)).
- Typical use case: n ≤ 10^5

```
n = int(input())
ar = []

for i in range(n):
  m = int(input())
  ar.append(m)

ar.sort() # O(n * log(n))
```

O(2^n)

Exponential time complexity.
Grows very rapidly.

Often requires memoization to avoid repeated computations and optimize.

- Subset generation, naive recursive Fibonacci...
- Backtracking without memoization.
- Typical use case: n ≤ 20

```
def fibonacci(n):
    if n <= 1: return n
    return fibonacci(n-1) + fibonacci(n-2)
    # O(2^n) without memorization
```

O(k · log(n))

K logarithmic operations time complexity.

- Extracting top k elements from a heap of size n
  - k extractions at O(log(n)) each
- k binary searches, heap push/pop k times...
- Typical use case: k ≪ n

O(N^2)

Quadratic time complexity.

- Nested loops (e.g., bubble sort, insertion sort).
- Many brute force solutions.
- All pairs of elements in a list.
- Typical use case: n ≤ 3000

```
for i in range(n):          # O(n²)
    for j in range(i, n):
        # Constant-time code
```

O(N!)

Factorial time.
Grows insanely rapidly.
Only solvable by computers for small n.

Often requires memoization to avoid repeated computations and optimize.

- Combinatorial problems, backtracking...
- Permutation generation, traveling salesman brute force
- Typical use case: n ≤ 12

Identify the best theoretical time complexity (BTTC)

The BTTC is the time complexity you cannot beat. For example:

- Finding the sum of all elements in an array: O(n) (must look at every element)
- Finding anagram groups: O(n · k) where n is the number of words and k is the maximum word length
- Finding islands in a matrix: O(n · m) where n is rows and m is columns

Knowing the BTTC prevents wasting time trying to find an impossible optimization. If your solution already matches the BTTC, then focus on doing less work within that complexity (single/multiple passes) and optimizing space complexity.

- Identify redundant work
  - Don't check conditions unnecessarily:
  - Avoid redundant conditions: `if not arr and len(arr) == 0` → `if not arr`
  - Check subconditions efficiently: `x < 5 and x < 10` → `x < 5`
  - Order checks by speed: `if slow() or fast()` → `if fast() or slow()`
  - Order checks by likelihood: `if unlikely() and likely()` → `if likely() and unlikely()`
- Cache repeated values:

```
# Instead of:
for i in range(len(array)):
    process(array[i])
```

```
# Do:
length = len(array)
for i in range(length):
    process(array[i])
```

- Use early termination:

```
# Instead of:
def contains_string(search_term, strings):
    result = False
    for string in strings:
        if string.lower() == search_term.lower():
            result = True
    return result
```

```
# Do:
def contains_string(search_term, strings):
    for string in strings:
        if string.lower() == search_term.lower():
            return True
    return False
```

- Minimize work inside loops:

```
# Instead of:
def contains_string(search_term, strings):
    for string in strings:
        if string.lower() == search_term.lower():
            return True
    return False
```

```
# Do:
def contains_string(search_term, strings):
    search_term_lowercase = search_term.lower()
    for string in strings:
        if string.lower() ==
search_term_lowercase:
            return True
    return False
```

- Avoid unnecessary computation:

```
def contains_string(search_term, strings):
    if not strings:  # Don't process if strings is empty
        return False
    search_term_lowercase = search_term.lower()
    for string in strings:
        if string.lower() == search_term_lowercase:
            return True
    return False
```

Space optimization techniques

Change data in-place:

- Modify the input array instead of creating new data structures
- Use the original array as a hash table (e.g., for marking presence)

Examples of common algorithm optimisation techniques

TECHNIQUE	BEFORE (NAÏVE)	AFTER (OPTIMIZED)	EXTRA SPACE	NOTES
Hashing (hash map/set)	$O(n^2)$ (nested loops)	O(n) (hash map lookup)	O(n)	Used in problems like Two Sum.
Binary search	O(n) (linear search)	O(log(n)) (binary search)	O(1)	Requires sorted array.
Sliding window	$O(n^2)$ (checking all subarrays)	O(n) (expanding window)	O(k) (k number of characters in the set)	Used for problems like longest substring.
Prefix sums	$O(n^2)$ (range sum queries)	O(1) (using precomputed sums), O(n) for computing sums	O(n)	Precompute for fast queries.
Bit manipulation	O(log(n)) (loop-based check)	O(1) (bitwise operations)	O(1)	Used for power-of-two checks.
Dynamic programming (memoization)	$O(2^n)$ (brute-force recursion)	O(n) (caching results)	O(n)	Used for Fibonacci, DP problems.
Union-Find (disjoint set)	$O(n^2)$ (brute-force merging)	O(α(n)) (path compression)	O(n)	Used in connected components, Kruskal's algorithm.
In-place modification	O(n) time, O(n) space	O(n) time, O(1) space	O(1)	Used in first missing positive, dutch national flag.

Hashing for faster lookups

Use case: When searching for elements or tracking frequency.

Key idea: Hash maps (`dict`) and hash sets (`set`) provide O(1) insert/search/delete on average.

Example: Two Sum problem, reduce $O(n^2)$ → O(n).

> *Given an array of integers* ***`nums`*** *and an integer* ***`target`****, return indices of the two numbers such that they add up to* ***`target`****. You may assume that each input would have exactly one solution, and you may not use the same element twice.*

https://leetcode.com/problems/two-sum/description/

```
def two_sum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i
```

✅ Uses O(n) extra space but avoids nested loops. ⚠️ Trading space for time.

Data structure complexities:

- Hash map lookup: O(1) average, O(n) worst case.
- Hash map insert: O(1) average, O(n) worst case.

Optimization insights:

- BTTC - since you need to examine each element at least once, the theoretical lower bound is O(n).
- By using a hash map, you eliminate reduntant work
    - Instead of checking every pair ($O(n^2)$), you do a single pass with O(1) lookups.
- Uses early termination by returning as soon as a valid pair is found.

Sorting + Binary search

Use case: When searching in a sorted array.

Key idea: Sorting first enables binary search (O(log(n))) instead of linear search (O(n)).

Example: Find target in a sorted array problem, reduce O(n) → O(log(n)).

> *Given a sorted array of integers* ***arr*** *and a target value* ***target****, return the index of* ***target*** *if it exists in the array, or -1 if it does not exist.*

https://leetcode.com/problems/binary-search/description/

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

✅ If the array is not previously sorted, sorting first takes O(n · log(n)) but allows O(log(n)) searches.

Data structure complexities:

- Binary search: O(log(n))
- Sorting (if required): O(n · log(n))

Optimization insights:

- Changing the approach completely from linear to binary search.
- Binary search is an example where the BTTC is actually lower than O(n) due to the sorted property.
- The algorithm uses early termination by returning as soon as the target is found.
- We avoid redundant work by eliminating half of the remaining search space in each iteration.

Sliding window

Use case: When working with subarrays or substrings.

Key idea: Expand and contract a window dynamically to avoid unnecessary computations.

Example: Longest substring without repeating characters, reduce $O(n^2) \rightarrow O(n)$.

> *Given a string* **s**, *find the length of the longest substring without repeating characters.*

https://leetcode.com/problems/longest-substring-without-repeating-characters/

```
def longest_unique_substring(s):
    seen = set()
    left = max_length = 0
    for right in range(len(s)):
        while s[right] in seen:
            seen.remove(s[left])
            left += 1
        seen.add(s[right])
        max_length = max(max_length, right - left + 1)
    return max_length
```

✅ Avoids checking every substring by sliding a window.

Data structure complexities:

- Set lookup: O(1)
- Set insert/delete: O(1)

Optimization insights:

- Recognizes overlapping and repeated computation in the naive approach (checking all substrings).
- Uses a sliding window to leverage previous computations.
- Uses a hash set for O(1) lookups to efficiently check if a character is already in the current window.
- Minimizes work inside loops by avoiding repeated scanning of the entire substring.

Prefix sums / Difference arrays

Use case: When working with range sum queries.

Key idea: Precompute prefix sums for O(1) range sum lookups.

Example: Range sum query, reduce $O(n^2)$ → O(n) preprocessing with O(1) queries.

> *Given an integer array* ***nums****, implement two functions:*
>
> ***prefix_sum(nums)*** *which precomputes a data structure*
>
> ***range_sum(prefix, left, right)*** *which calculates the sum of elements from index* ***left*** *to index* ***right*** *(inclusive)*

https://leetcode.com/problems/range-sum-query-immutable/

```
def prefix_sum(arr):
    prefix = [0] * (len(arr) + 1)
    for i in range(len(arr)):
        prefix[i+1] = prefix[i] + arr[i]
    return prefix

def range_sum(prefix, left, right):
    return prefix[right+1] - prefix[left]
```

✅ Preprocessing O(n), but each query is O(1).

Data structure complexities:

- Array lookup: O(1)

Optimization insights:

- Identifying overlapping and repeated computation, since a naive approach would recalculate sums multiple times.
- Preprocessing allows for avoiding redundant calculations during queries.
- Once preprocessed, queries become O(1) which is the BTTC for range sum queries.
- The pattern of "do more work upfront to save time later" is very common in optimization.

Bit manipulation

Use case: When working with numbers and optimizations involving powers of two.

Key idea: Use bitwise operations for faster calculations.

Example: Check if a number is a power of two, reduce O(log(n)) → O(1)

> *Given an integer* ***n****, return* ***true*** *if it is a power of two. Otherwise, return* ***false****.*

https://leetcode.com/problems/power-of-two/description/

```
def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0
```

✅ Avoids looping by using bitwise tricks.

Data structure complexities:

- Bitwise operations: O(1)

Optimization insights:

- Uses mathematical properties, rather than algorithmic optimizations.
- Powers of two have exactly one bit set, and this property lets us avoid iteration entirely.
- The solution is both time and space optimal.

Dynamic programming

Use case: When solving problems with overlapping subproblems.

Key idea: Store computed results using memoization (top-down recursion) or tabulation (bottom-up iteration).

Example: Fibonacci sequence, reduce $O(2^n)$ → O(n) with memorization.

> *The Fibonacci numbers form a sequence where each number is the sum of the two preceding ones, starting from 0 and 1. Given **n**, calculate the nth Fibonacci number.*

https://leetcode.com/problems/fibonacci-number/

```
def fib(n, memo=None):
    if memo is None:  # Avoid mutable default argument issues
        memo = {}
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

✅ Stores results instead of recomputing values.

Data structure complexities:

- Dictionary lookup: O(1)

Optimization insights:

- Identifying overlapping and repeated computation.
- Uses memoization to avoid recalculating the same Fibonacci numbers multiple times.
- Demonstrates how to convert an exponential algorithm to a linear one by caching results.
- Only compute values when needed (top-down approach).

Union-find (disjoint set)

Use case: When handling connected components in a graph.

Key idea: Union-find optimizes union and find operations using path compression.

Example: Union-find for connected components, reduces $O(n^2)$ to $O(\alpha(n))$.

> *Design a data structure that keeps track of a set of elements partitioned into a number of disjoint subsets. It should support the following operations:*
>
> ***find(x)**: Determine which subset element **x** belongs to*
>
> ***union(x, y)**: Join two subsets containing elements **x** and **y** into a single subset*

https://leetcode.com/problems/number-of-provinces/

https://leetcode.com/problems/redundant-connection/description/

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [1] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x]) # Path Compression
        return self.parent[x]

    def union(self, x, y):
        rootX, rootY = self.find(x), self.find(y)
        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1
```

✅ Works in nearly constant time $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, and is basically $O(1)$.

Data structure complexities:

- Union: $O(\alpha(n))$
- Find: $O(\alpha(n))$

Optimization insights:

- Change of data structure from adjacency lists/matrices to a specialized disjoint-set structure.
- Uses path compression to flatten the tree during find operations.
- Uses union by rank to minimize tree height.
- Shows how specialized data structures can provide dramatic improvements for specific operations.

First missing positive

Use case: when you need to optimize space complexity to O(1).

Key idea: use the input array itself as a hash table for marking presence.

Example: First missing positive, reduce O(n) space to O(1) space.

> *Given an unsorted integer array* ***nums****, find the smallest missing positive integer.*

https://leetcode.com/problems/first-missing-positive/description/

```
def first_missing_positive(nums):
    n = len(nums)

    # Step 1: Replace negative numbers with n+1
    for i in range(n):
        if nums[i] <= 0:
            nums[i] = n + 1

    # Step 2: Mark presence by negating values
    for i in range(n):
        num = abs(nums[i])
        if 1 <= num <= n:
            nums[num - 1] = -abs(nums[num - 1])

    # Step 3: Find first positive index
    for i in range(n):
        if nums[i] > 0:
            return i + 1

    return n + 1
```

✅ Uses the array itself to mark the presence of values without using extra space.

Data structure complexities:

- Different data structures have different space requirements.

Optimization insights:

- The algorithm uses the original array as a hash table, avoiding O(n) extra space.
- The algorithm makes three passes through the array, but still achieves O(n) time complexity.
- Uses sign bits to incidate the presence of a specific value (1 to n).
- Achieves BTTC O(n) time, while maintaining O(1) space complexity.

Technical Interview Guide

Problem solving framework

7 step approach

1. **Listen** - Pay very close attention to all the information in the problem description, as you'll likely need all of it for an optimal solution.
    a. Clarify assumptions before jumping to coding
    b. Ask about time/space complexity requirements or constraints
    c. Understand what makes a valid input and expected output
2. **Example** - Pay very close attention to all information in the problem description, as you'll likely need all of it for an optimal solution.
    a. Clarify assumptions before jumping to coding
    b. Ask about time/space complexity requirements or constraints
    c. Understand what makes a valid input and expected output
3. **Brute force** - Develop a brute-force solution first. State the naive algorithm and its runtime before optimizing.
    a. Don't worry about efficiency initially
    b. Focus on correctness first
4. **Optimize** - Walk through your brute force with BUD optimization or try these approaches:
    a. Look for unused information (you usually need all information in a problem)
    b. Solve manually, then reverse-engineer your thought process
    c. Solve "incorrectly" and analyze what fails
    d. Make time vs. space tradeoffs (hash tables are especially useful)
    e. Choose more appropriate data structures/algorithms
    f. Use more memory to improve speed when needed
5. **Walk through** - Before coding, walk through your approach in detail to ensure you understand each step.
    a. Trace through the algorithm with a simple example
    b. Look for off-by-one errors
    c. Confirm your approach handles all edge cases
6. **Implement** - Write beautiful, modular code from the beginning and refactor as needed.
    a. Write pure functions as much as possible
    b. Avoid mutating parameters passed into functions
    c. Avoid relying on or mutating global variables
    d. Balance functional and imperative approaches:
        i. Functional programming → More readable but expensive in space (repeated object allocation)
        ii. Imperative code → Faster because you operate on existing objects
7. **Test** - Test in this order:
    a. Always validate input first (invalid/empty/negative/different types)
    b. Conceptual test (code review style)
    c. Unusual or non-standard code
    d. Hot spots (arithmetic, null nodes)
    e. Small test cases (faster and just as effective)
    f. Special cases and edge cases
    g. Check for off-by-one errors
    h. Use your examples to validate the solution
    i. Fix bugs when found carefully

Coding Skills

BUD Optimization

- **B**ottlenecks
- **U**nnecessary Work
- **D**uplicated Work

5 approaches to problem solving

- BUD: Look for bottlenecks, unnecessary work, duplicated work
- DIY: Do It Yourself (solve manually)
- Simplify & Generalize: Solve a simpler version
- Base Case & Build: Solve for base cases then build
- Data Structure Brainstorm: Try various data structures

Speed improvement principles

- Best Conceivable Runtime (BCR) is the runtime you know you can't beat (e.g., O(|A|+|B|) for computing the intersection of two sets).
    - Choose a more appropriate data structure or algorithm
    - Use more memory to reduce computation time
    - Precompute and cache results when possible

What good coding looks like

- Correct
- Efficient
- Simple
- Readable
- Maintainable

Beautiful code guidelines

- Modularized code: Use good coding style with functions for reusable operations. Don't waste time writing boilerplate initialization code during interviews - you can use placeholder functions like `initIncrementalMatrix(int size)` and explain later if needed.
- Error checks: Find a balance - add `todo` comments and verbally explain what you'd test, since different interviewers have different expectations about error handling.
- Use appropriate data structures: When a function needs to return complex data (like start/end points), use custom objects/classes (like `StartEndPair` or `Range`) rather than just returning a 2D array. You can sketch the class structure without implementing all details.
- Meaningful variable names:
    - Avoid single-letter variables except where conventional (like `i` and `j` in simple loops)
    - Use descriptive names for important variables
      (e.g., `startChild` instead of `int i = startOfChild(array)`)
    - For longer variable names, you can define them first
      then explain you'll abbreviate them (e.g., `startChild` → `sc`)

Behavioral Interview

Interview preparation grid

Listing major aspects of the resume (including projects, work, activities); 1-3 projects that can be talked about in detail.

	Project 1	Project 2	Project 3
Challenges			
Mistakes/Failures			
Enjoyed			
Leadership			
Conflicts			
What you'd change			

S.A.R. technique

- Nugget - Describe what the response will be about.
- Situation - Outline the situation.
    - Present a challenge and situation in which you found yourself in.
- Action - Explain the actions you took.
    - What did you do? What you did, why you did it, and what the alternatives were.
- Result - Result of said action.
    - What was the outcome of your actions? What did you achieve through your actions and what did you learn? What steps did you take after to improve?

	Nugget	Situation	Action(s)	Results	What it says
Story 1...					
Story 2...					
Story 3...					
Story 4...					

Self-introduction

"Tell me about your journey into tech. How did you get interested in coding, and why was web development (or replace with other job-specific skills) a good fit for you? How is that applicable to our role or company goals?" It is probably not a good idea to spend valuable time talking about things which aren't relevant to the job!

- Current role (headline only)
- College
- Post college & afterwards
- Current role (details)
- Outside of work
- Wrap up

Elevator pitch it: short (limited time), direct (get to the point), and attention-grabbing (most attractive ideas or traits). The elevator recipe:

- Who you are, past experiences (school/work), and what you do.
- KISS (Keep It Simple and Sweet) - executive summary example: "I reverse engineered X game by decrypting Y packet to predict Z."
- Why would they want you, why you would be a good hire - Is your experience relevant? Have you used the same tech stack? What unique talents do you have to contribute to the company?

Common evaluated values

1. **Motivation** - What drives you? Ideal candidates are self-motivated, passionate about technologies and products that have a real impact.
    a. What project are you most proud of and why?
    b. Tell me about a recent day working that was really great and/or fun.
2. **Ability to be proactive** - Are you able to take initiative? Given a difficult problem, are you able to figure out how to get it done and execute on it?
    a. Tell me about a time when you wanted to change something that was outside of your regular scope of work.
    b. Tell me about a time you had to make a fast decision and live with the results.
3. **Ability to work in an unstructured environment** - How well are you able to take ownership in ambiguous situations? Or do you rely on others to be told what to do?
    a. How do you decide what to work on next?
    b. Tell me about a project or task that was ambiguous or underspecified.
4. **Perseverance** - Are you able to push through difficult problems or blockers?
    a. Tell me about a time when you needed to overcome external obstacles to complete a task or project.
    b. Tell me about a time a project took longer as expected.
5. **Conflict resolution & Empathy** - How well are you able to handle and work through challenging relationships? How well are you able to see things from the perspective of others and understand your motivations?
    a. Tell me about a person or team who you found the most challenging to work with.
    b. Tell me about a time you disagreed with a coworker.
    c. Tell me about a situation where two teams couldn't agree on a path forward.
6. **Growth** - How well do you understand your strengths, weaknesses and growth areas? Are you making a continued effort to grow?
    a. Describe a situation when you made a mistake, and what you learned from it.
    b. Tell me about some constructive feedback you received from a manager or a peer.
    c. Tell me about a skill set that you observed in a peer or mentor that you want to develop in the next six months.
7. **Communication** - Are you able to clearly communicate your stories during the interview?

Questions

Commonly asked behavioral questions

1. Why do you want to work for X company?
2. Why do you want to leave your current/last company?
3. What are you looking for in your next role?
4. Tell me about a time when you had a conflict with a co-worker.
5. Tell me about a time in which you had a conflict and needed to influence somebody.
6. What project are you currently working on?
7. What is the most challenging aspect of your current project?
8. What was the most difficult bug that you fixed in the past 6 months?
9. How do you tackle challenges? Name a difficult challenge you faced while working on a project, how you overcame it, and what you learned.
10. What are you excited about?
11. What frustrates you?
12. Imagine it is your first day here at the company. What do you want to work on? What features would you improve on?
13. What are the most interesting projects you have worked on and how might they be relevant to this company's environment?
14. Tell me about a time you had a disagreement with your manager.
15. Talk about a project you are most passionate about, or one where you did your best work.
16. What does your best day of work look like?
17. What is something that you had to push for in your previous projects?
18. What is the most constructive feedback you have received in your career?
19. What is something you had to persevere at for multiple months?
20. Tell me about a time you met a tight deadline.
21. If this were your first annual review with our company, what would I be telling you right now?
22. Time management has become a necessary factor in productivity. Give an example of a time-management skill you've learned and applied at work.
23. Tell me about a problem you've had getting along with a work associate.
24. What aspects of your work are most often criticized?
25. How have you handled criticism of your work?
26. What strengths do you think are most important for your job position?
27. What words would your colleagues use to describe you?
28. What would you hope to achieve in the first six months after being hired?
29. Tell me why you will be a good fit for the position.

Questions to ask at the end of interviews

Technical work:

1. What are the engineering challenges that the company/team is facing?
2. What has been the worst technical blunder that has happened in the recent past? How did you guys deal with it? What changes were implemented afterwards to make sure it didn't happen again?
3. What is the most costly technical decision made early on that the company is living with now?
4. What is the most fulfilling/exciting/technically complex project that you've worked on here so far?
5. I do/don't have experience in domain X. How important is this for me to be able to succeed?
6. How do you evaluate new technologies? Who makes the final decisions?
7. How do you know what to work on each day?
8. How would you describe your engineering culture?
9. How has your role changed since joining the company?
10. What is your stack? What is the rationale for/story behind this specific stack?
11. Do you tend to roll your own solutions more often or rely on third party tools? What's the rationale in a specific case?
12. How does the engineering team balance resources between feature requests and engineering maintenance?
13. What do you measure? What are your most important product metrics?
14. How often have you moved teams? What made you join the team you're on right now? If you wanted to move teams, what would need to happen?

15. What resources does the company have for new hires to study its product and processes? Are there specifications, requirements, documentation?
16. How do you think my expertise would be relevant to this team? What unique value can I add?

The role:

1. What qualities do you look out for when hiring for this role?
2. What would be the most important problem you would want me to solve if I joined your team?
3. What does a typical day look like in this role?
4. What are the strengths and weaknesses of the current team? What is being done to improve upon the weaknesses?
5. What resources does the company have for new hires to study its product and processes? Are there specifications, requirements, documentation?
6. What would I work on if I joined this team and who would I work most closely with?

Culture and welfare:

1. What is the most frustrating part about working here?
2. What is unique about working at this company that you have not experienced elsewhere?
3. What is something you wish were different about your job?
4. How is individual performance measured?
5. What do you like about working here?
6. What is your policy on working from home/remotely?
7. What does the company do to nurture and train its employees?
8. Does the company culture encourage entrepreneurship and creativity? Could you give me any specific examples?
9. Leadership and management:
10. How do you train/ramp up engineers who are new to the team?
11. What does success look like for your team/project?
12. What are the strengths and weaknesses of the current team? What is being done to improve upon the weaknesses?
13. Can you tell me about a time you resolved an interpersonal conflict?
14. How did you become a manager?
15. How do your engineers know what to work on each day?
16. What is your team's biggest challenge right now?
17. How do you measure individual performance?
18. How often are 1:1s conducted?
19. What is the current team composition like?
20. What opportunities are available to switch roles? How does this work?
21. Two senior team members disagree over a technical issue. How do you handle it?
22. Have you managed a poor performer at some point in your career before? What did you do and how did it work?
23. Where do you spend more of your time, high performers or low performers?
24. Sometimes there's a trade-off between what's best for one of your team members and what's best for the team. Give an example of how you handled this and why.
25. Give an example of a time you faced a difficult mentoring/coaching challenge. What did you do and why?
26. What is your management philosophy?
27. What is the role of data and metrics in managing a team like ours?
28. What role does the manager play in making technical decisions?
29. What is an example of a change you have made in the team that improved the team?
30. What would be the most important problem you would want me to solve if I joined your team?
31. What opportunities for growth will your team provide?
32. What would I work on if I joined this team and who would I work most closely with?
33. Company direction:
34. How does the company decide on what to work on next?
35. What assurance do you have that this company will be successful?
36. Which companies are your main competitors and what differentiates your company?
37. What are your highest priorities right now? For example, new features, new products, solidifying existing code, reducing operations overhead?

Data Structures

Array

Arrays store elements of the same type in contiguous memory locations, allowing for constant-time access via indices.

In Python, `lists` are dynamic arrays that can store different data types and automatically resize.

- Advantages: O(1) access by index; efficient iteration, cache-friendly memory layout.
- Disadvantages: O(n) insertions/deletions in the middle; fixed size in most languages (except Python).

Terminology:

- Subarray - Contiguous slice of an array (e.g., in `[2,3,6,1,5,4]`, `[3,6,1]` is a subarray).
- Subsequence - Elements in original order but not necessarily contiguous ((e.g., `[3,1,5]` is a subsequence)

Interview tips:

- Clarify if array is sorted (enables binary search)
- Ask about duplicates and how they should be handled
- Consider in-place vs. extra space solutions
- Watch for off-by-one errors and array bounds
- Consider pre-processing the array for optimisation

Edge cases:

- Empty array
- Single-element array
- Array with duplicates
- Array with all identical elements
- Sorted vs. unsorted considerations

Operation	Time	Space	Notes
Access	O(1)	O(1)	Direct index calculation.
Search (unsorted)	O(n)	O(1)	Linear scan required.
Search (sorted)	O(n · log(n))	O(1)	Binary search applicable.
Insert (middle)	O(n)	O(1)	Requires shifting elements.
Insert (end)	O(1) *	O(1)	* Amortised for dynamic arrays.
Delete (middle)	O(n)	O(1)	Requires shifting elements.
Delete (end)	O(1)	O(1)	No shifting needed.

Common techniques

Sliding window

Maintain a "window" using two pointers that typically move in the same direction.

When to use: Finding subarrays with certain properties (sum constraints, no duplicates...).

```
# Template for sliding window
def sliding_window(arr):
    start = 0
    result = 0  # or other initial value

    for end in range(len(arr)):
        # Expand window by including element at end pointer
        # process logic for expansion

        # Contract window when invalid condition is met
        while invalid_condition:
            # process logic for contraction
            start += 1

        # Update result if needed
        result = max(result, end - start + 1)  # or other calculation

    return result

# Example: Find longest subarray with sum <= k
def longest_subarray_sum_constraint(arr, k):
    start = 0
    curr_sum = 0
    max_length = 0

    for end in range(len(arr)):
        curr_sum += arr[end]

        while start <= end and curr_sum > k:
            curr_sum -= arr[start]
            start += 1

        max_length = max(max_length, end - start + 1)

    return max_length
```

Problems:

1. https://leetcode.com/problems/longest-substring-without-repeating-characters/
2. https://leetcode.com/problems/minimum-size-subarray-sum/
3. https://leetcode.com/problems/minimum-window-substring/description/

Two pointers

Use two pointers that may move toward or away from each other. A more general version of sliding window, which can also be used on different arrays.

When to use: Pair finding, partitioning, palindrome detection.

```
# Template for two pointers from opposite ends
def two_pointers_opposite(arr):
    left, right = 0, len(arr) - 1
    result = 0  # or other initial value

    while left < right:
        # Process current elements
        current = process(arr[left], arr[right])

        # Update result
        result = max(result, current)  # or other calculation

        # Move pointers based on condition
        if condition:
            left += 1
        else:
            right -= 1

    return result

# Example: Two Sum II (input array is sorted)
def two_sum_sorted(numbers, target):
    left, right = 0, len(numbers) - 1

    while left < right:
        current_sum = numbers[left] + numbers[right]

        if current_sum == target:
            return [left + 1, right + 1]  # 1-indexed
        elif current_sum < target:
            left += 1
        else:
            right -= 1

    return [-1, -1]  # No solution
```

Problems:

1. https://leetcode.com/problems/sort-colors/
2. https://leetcode.com/problems/palindromic-substrings/
3. https://leetcode.com/problems/merge-sorted-array/

Traversing from the right

Traversing the array starting from the right, instead of the conventional approach from the left.

When to use: Problems where the right side has an impact on the left side, monotonic stack problems, next greater element problems.

```
# Template for right to left traversal
def traverse_right_to_left(arr):
    n = len(arr)
    result = [0] * n

    for i in range(n - 1, -1, -1):
        # Process current element
        # Update result based on elements to the right

    return result

# Example: Daily Temperatures - next warmer day
def daily_temperatures(temperatures):
    n = len(temperatures)
    result = [0] * n
    stack = []

    for i in range(n - 1, -1, -1):
        while stack and temperatures[stack[-1]] <= temperatures[i]:
            stack.pop()

        if stack:
            result[i] = stack[-1] - i

        stack.append(i)

    return result
```

Problems:

1. https://leetcode.com/problems/daily-temperatures/
2. https://leetcode.com/problems/number-of-visible-people-in-a-queue/

Sorting the array

If an array is already completely/partially sorted, some kind of binary search should be possible - usually meaning there is a solution faster than O(n).

When to use: When order matters, interval problems, finding optimal arrangements.

```
# Example: Merge Intervals
def merge_intervals(intervals):
    if not intervals:
        return []

    # Sort by start time
    intervals.sort(key=lambda x: x[0])

    merged = [intervals[0]]

    for current in intervals[1:]:
        # Get the last added interval
        previous = merged[-1]

        # If current interval overlaps with previous
        if current[0] <= previous[1]:
            # Merge them by updating end time
            previous[1] = max(previous[1], current[1])
        else:
            # Add as a new interval
            merged.append(current)

    return merged
```

Problems:

1. https://leetcode.com/problems/merge-intervals/description/
2. https://leetcode.com/problems/non-overlapping-intervals/description/

Precomputation

Precompute cumulative sums or multiplications to enable O(1) subarray calculations.

When to use: Subarray sum problems, range queries.

```
# Prefix sum computation
def prefix_sum(arr):
    n = len(arr)
    prefix = [0] * (n + 1)

    for i in range(n):
        prefix[i + 1] = prefix[i] + arr[i]

    return prefix

# Query sum of subarray [i, j] inclusive
def query_sum(prefix, i, j):
    return prefix[j + 1] - prefix[i]

# Example: Find subarrays with sum equal to target
def subarrays_with_target_sum(nums, target):
    prefix_sum = {0: 1}  # Sum: count
    curr_sum = 0
    count = 0

    for num in nums:
        curr_sum += num
        if curr_sum - target in prefix_sum:
            count += prefix_sum[curr_sum - target]

        prefix_sum[curr_sum] = prefix_sum.get(curr_sum, 0) + 1

    return count
```

Problems:

1. https://leetcode.com/problems/product-of-array-except-self/description/
2. https://leetcode.com/problems/minimum-size-subarray-sum/description/
3. https://leetcode.com/problem-list/prefix-sum/

Index as hash key

Use array indices to mark visited elements (when values are in range 0 to n-1). Given a sequence and request for O(1) space, the array itself can be used as a hash table.

When to use: Finding duplicates, missing numbers in a range.

```
# Example: Find duplicate in array with values 1 to n
def find_duplicate(nums):
    n = len(nums)

    for i in range(n):
        # Get the absolute value
        index = abs(nums[i])

        # If already negative, we've seen this index before
        if nums[index] < 0:
            return index

        # Mark as visited by making negative
        nums[index] = -nums[index]

    # Restore array (optional)
    for i in range(n):
        nums[i] = abs(nums[i])

    return -1  # No duplicate found
```

Problems:

1. https://leetcode.com/problems/first-missing-positive/description/
2. https://leetcode.com/problems/daily-temperatures/description/

Kadane's algorithm

Dynamic programming approach to find maximum subarray sum.

When to use: Maximum subarray problems, contiguous sequence optimization.

```
# Basic Kadane's algorithm
def kadane(arr):
    if not arr:
        return 0

    current_sum = max_sum = arr[0]

    for num in arr[1:]:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)

    return max_sum

# Extended version with subarray bounds
def kadane_with_indices(arr):
    if not arr:
        return 0, -1, -1

    current_sum = max_sum = arr[0]
    current_start = max_start = max_end = 0

    for i in range(1, len(arr)):
        if arr[i] > current_sum + arr[i]:
            current_sum = arr[i]
            current_start = i
        else:
            current_sum = current_sum + arr[i]

        if current_sum > max_sum:
            max_sum = current_sum
            max_start = current_start
            max_end = i

    return max_sum, max_start, max_end
```

Problems:

1. https://leetcode.com/problems/maximum-subarray/
2. https://leetcode.com/problems/maximum-product-subarray/

String

A string is a sequence of characters, essentially an array of characters. Strings are one of the most common data types in programming and frequently appear in interview questions.

Common data structures and algorithms for strings:

- Trie/Prefix tree - Efficient for prefix searches and autocomplete.
- Suffix tree - Optimized for substring searches and pattern matching.
- Robin-Karp - Uses rolling hash for efficient substring searching (O(n + m) average case).
- Knuth-Morris-Pratt (KMP) - Optimized substring search by avoiding unnecessary comparisons (O(n + m) worst case).

Interview tips:

- Clarify input character set (ASCII, Unicode, lowercase only, etc.)
- Ask about case sensitivity requirements
- Consider space usage - can you use O(1) extra space?
- Watch for off-by-one errors in string operations
- Consider preprocessing input (e.g., removing spaces for palindromes)

Edge cases:

- Empty string
- Single character string
- String with all identical characters
- String with only unique characters
- Case sensitivity considerations
- Special characters and spaces

Operation	Time	Space	Notes
Access	O(1)	O(1)	Random access by index.
Search (linear)	O(n)	O(1)	Scan for a character.
Find substring	O(n · m)	O(1)	Naive approach (n = string length, m = pattern length).
Concatenation	O(n + m)	O(n + m)	Creates new string.
Insert/Remove	O(n)	O(n)	Requires creating new string in most languages.
Slice	O(m)	O(m)	m = slice length
Split	O(n + m)	O(n + m)	n = string length, m = number of splits
Strip	O(n)	O(n)	Removing leading/trailing whitespace.

Common techniques

Counting characters

Use a hash table or counter in Python to track frequency of characters. Common mistake is to say the space complexity of the counter is O(n), but it is O(1), because there is a limited number of possible characters, usually. Counting takes O(n) time, and O(k) space, where k is the number of unique characters.

When to use: Frequency analysis, anagram detection, permutation checks.

```
from collections import Counter


# Input string
s = "abracadabra"

# Using a regular dictionary
char_count = {}
for char in s:
    char_count[char] = char_count.get(char, 0) + 1

# Using Counter
counter = Counter(s)

# Result (for both methods):
# {'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}
```

Bit manipulation for unique characters

Using bits to track character presence for strings with unique characters. Use a 26-bit bitmask to indicate which lower case latin characters are inside the string. To determine if two strings have common characters, perform & on the two bitmasks. If the result is non-zero (mask_a & mask_b > 0), then the two strings have common characters. This is O(n) time, with O(1) space.

When to use: Unique character detection, character set intersection, smaller memory footprint.

```
def count_unique_chars(s):
    mask = 0
    for char in s:
        # Set the bit corresponding to this character
        mask |= (1 << (ord(char) - ord('a')))

    # Count the number of set bits
    return bin(mask).count('1')

# Usage
print(count_unique_chars("abcab"))  # Output: 3
```

Anagram detection

Methods to determine if two strings are anagrams (same characters in different order, while using all of the original letters only once).

When to use: Group anagrams, permutation checking, cryptography.

```
# Method 1: Sorting
def are_anagrams_sort(s1, s2):
    return sorted(s1) == sorted(s2)

# Example
print(are_anagrams_sort("listen", "silent"))  # True

# Method 2: Character counting
from collections import Counter

def are_anagrams_count(s1, s2):
    return Counter(s1) == Counter(s2)

# Example
print(are_anagrams_count("anagram", "nagaram"))  # True

# Method 3: Prime numbers
def are_anagrams_prime(s1, s2):
    if len(s1) != len(s2):
        return False

    primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89,
97, 101]

    def string_to_value(s):
        value = 1
        for char in s:
            value *= primes[ord(char) - ord('a')]
        return value

    return string_to_value(s1) == string_to_value(s2)

# Example
print(are_anagrams_prime("cat", "act"))  # True
```

Problems:

1. https://leetcode.com/problems/group-anagrams/

Palindrome detection

Methods to detect if a string reads the same forward and backward.

When to use: Finding palindromic substrings, checking if a string is a palindrome.

```
# Method 1: Reverse and compare
def is_palindrome_reverse(s):
    return s == s[::-1]

# Example
print(is_palindrome_reverse("racecar"))  # True
print(is_palindrome_reverse("hello"))    # False

# Method 2: Two pointers moving inward
def is_palindrome_inward(s):
    left, right = 0, len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True

# Example
print(is_palindrome_inward("level"))  # True
print(is_palindrome_inward("python")) # False

# Method 3: Expand from center
def is_palindrome_outward(s):
    def expand_around_center(left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return right - left - 1 == len(s)

    n = len(s)
    # Check odd length palindromes
    if expand_around_center(n // 2, n // 2):
        return True
    # Check even length palindromes
    if n > 1 and expand_around_center(n // 2 - 1, n // 2):
        return True
    return False
```

Problems:

1. https://leetcode.com/problems/longest-palindromic-substring/
2. https://leetcode.com/problems/longest-palindromic-subsequence/description/

Tree

Trees are hierarchical data structures comprised of nodes connected by edges. Each node in the tree can be connected to many children, but must be connected to exactly one parent, except for the root node, which has no parent. Trees are undirected, connected acyclic graphs with no cycles or loops.

Key property: Each node can be viewed as the root of its own subtree, making recursion an ideal technique for traversal.

Common use cases: Representing hierarchical data (file systems, JSON, HTML documents), efficient search (BST), priority systems (heaps).

Terminology:

- Node: Basic unit containing data and references to child nodes.
- Root: Topmost node with no parent.
- Leaf: Node with no children.
- Neighbor: Parent or child of a node.
- Ancestor: Node reachable by traversing up the parent chain.
- Descendant: Any node in a node's subtree.
- Degree: Number of children of a node.
- Degree of a tree: Maximum degree of any node.
- Distance: Number of edges along shortest path between two nodes.
- Level/Depth: Number of edges from root to the node.
- Height: Maximum depth of any node.
- Width: Number of nodes at a specific level.
- Binary tree: Each node has at most 2 children (left and right).
- Complete binary tree: Every level is filled except possibly the last, where all nodes are as far left as possible.
- Balanced binary tree: Left and right subtree heights differ by at most 1 for all nodes.
- Full Binary Tree: Every node has either 0 or 2 children.
- Perfect Binary Tree: All interior nodes have 2 children and all leaves are at the same level.

Interview tips:

- Clarify tree type (binary, BST, n-ary)
- Ask about balancing requirements
- Consider both recursive and iterative approaches
- For BST problems, remember the ordering property
- Watch for subtree problems that can leverage recursion
- Be careful with null checks and boundary conditions
- Binary Search Tree (BST)
- Special binary tree with the property - for any node N:
- All nodes in N's left subtree have values < N's value
- All nodes in N's right subtree have values > N's value
- In-order traversal yields elements in sorted order
- Operations typically run in O(log(n)) time (if balanced)

Edge cases:

- Empty tree (null root)
- Single node tree
- Two-node tree
- Skewed tree (degenerates to linked list)
- Complete vs. incomplete last level
- Duplicate values (especially in BSTs)

Operation	Time		Space	Notes
	Average	Worst		
Access	O(log(n))	O(n)	O(1)	
Search	O(log(n))	O(n)	O(1)	
Insert	O(log(n))	O(n)	O(1)	
Delete	O(log(n))	O(n)	O(1)	
DFS traversal	O(n)		O(h)	All variants, where h is the tree height.
BFS traversal	O(n)		O(w)	Level order, where w is the max width of the tree.

Traversal methods

To uniquely serialize (store or transmit) a binary tree, you need in-order traversal + either pre-order or post-order. This ensures that the tree can be reconstructed exactly as it was.

Depth-first traversals

- In-order traversal: Left → Root → Right
- Pre-order traversal: Root → Left → Right
- Post-order traversal: Left → Right → Root

Breadth-first traversal

- Level-order traversal: All nodes at level 0, then all on level 1, and so on

Common techniques

Recursion

Most natural approach for tree problems due to their recursive structure.

When to use: Tree traversal, calculating properties (height, count), path finding.

```
# Example: Calculate height of a binary tree
def tree_height(root):
    if not root:
        return 0
    return max(tree_height(root.left), tree_height(root.right)) + 1

# Example: Preorder traversal
def preorder_traversal(root):
    result = []

    def dfs(node):
        if not node:
            return

        # Visit the root first (preorder)
        result.append(node.val)

        # Then the left subtree
        dfs(node.left)

        # Then the right subtree
        dfs(node.right)

    dfs(root)
    return result
```

Problems:

1. https://leetcode.com/problems/maximum-depth-of-binary-tree/
2. https://leetcode.com/problems/diameter-of-binary-tree/

Level-order processing

Use BFS when nodes need to be processed level by level.

When to use: Level-based problems, finding shortest paths, zigzag traversal.

```
# Example: Level order traversal
from collections import deque

def level_order_traversal(root):
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        level = []

        for _ in range(level_size):
            node = queue.popleft()
            level.append(node.val)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(level)

    return result
```

Problems:

1. https://leetcode.com/problems/binary-tree-level-order-traversal/
2. https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/

Parent-Child relationship

Track relationships between nodes to solve ancestry problems. DFS/BFS traversal is required to populate the parent map before solving the problem.

When to use: Lowest common ancestor, path finding between nodes.

```
# Example: Find lowest common ancestor
def lowest_common_ancestor(root, p, q):
    # Maps each node to its parent
    parent_map = {}

    def dfs(node, parent=None):
        if not node:
            return

        parent_map[node] = parent
        dfs(node.left, node)
        dfs(node.right, node)

    dfs(root)

    # Get path from p to root
    p_ancestors = set()
    while p:
        p_ancestors.add(p)
        p = parent_map.get(p)

    # Check each ancestor of q until we find one in p's ancestors
    while q:
        if q in p_ancestors:
            return q
        q = parent_map.get(q)

    return None
```

Problems:

1. https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/
2. https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/

Tree construction

Building trees from traversal data or other specifications. In-order traversal is necessary for determining left and right subtree boundaries.

When to use: Deserializing trees, constructing from traversal results.

```
# Example: Construct binary tree from preorder and inorder traversal
def build_tree(preorder, inorder):
    if not preorder or not inorder:
        return None

    # Root is first element in preorder
    root_val = preorder[0]
    root = TreeNode(root_val)

    # Find position of root in inorder
    inorder_idx = inorder.index(root_val)

    # Recursively build left and right subtrees
    root.left = build_tree(
        preorder[1:inorder_idx+1],
        inorder[:inorder_idx]
    )

    root.right = build_tree(
        preorder[inorder_idx+1:],
        inorder[inorder_idx+1:]
    )

    return root
```

Problems:

1. https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/
2. https://leetcode.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/

Bottom-Up recursion

Process child nodes before making decisions at current node.

When to use: Maximum path sum, balanced tree verification.

```
# Example: Check if a binary tree is balanced
def is_balanced(root):
    def height(node):
        if not node:
            return 0

        # Get heights of left and right subtrees
        left_height = height(node.left)
        if left_height == -1:
            return -1

        right_height = height(node.right)
        if right_height == -1:
            return -1

        # If height difference is more than 1, tree is not balanced
        if abs(left_height - right_height) > 1:
            return -1

        # Return height of current subtree
        return max(left_height, right_height) + 1

    return height(root) != -1
```

Problems:

1. https://leetcode.com/problems/balanced-binary-tree/
2. https://leetcode.com/problems/binary-tree-maximum-path-sum/description/

Graph

A graph is a structure containing sets of objects (nodes or vertices) where there can be edges between these nodes/vertices. Graphs are versatile data structures that can model various relationships and connections.

Common data structures and algorithms for graphs:

- Adjacency matrix - Space-efficient for dense graphs, $O(V^2)$ space.
- Adjacency list - Efficient for sparse graphs, O(V + E) space.
- Depth-First Search (DFS) - Explores deep paths before backtracking.
- Breadth-First Search (BFS) - Explores all neighbors before moving deeper.
- Topological Sort - Linear ordering of vertices where for every edge (u,v), u comes before v.
- Dijkstra's algorithm - Finds shortest paths in weighted graphs without negative edges.

Terminology:

- Node/Vertex: Basic unit in a graph that can hold data.
- Edge: Connection between two vertices, can be directed or undirected.
- Adjacent/Neighbor: Two vertices connected by an edge.
- Path: Sequence of vertices connected by edges.
- Cycle: Path that starts and ends at the same vertex.
- Connected Graph: Graph where there's a path between
- Connected Graph: Graph where there's a path between every pair of vertices.
- Disconnected Graph: Graph with at least one pair of vertices with no path between them.
- Connected Component: Maximal connected subgraph.
- Degree: Number of edges connected to a vertex.
- In-degree: Number of incoming edges to a vertex (directed graphs).
- Out-degree: Number of outgoing edges from a vertex (directed graphs).
- Weighted Graph: Graph where edges have associated values/weights.
- Directed Graph (Digraph): Graph where edges have direction.
- Undirected Graph: Graph where edges have no direction.
- Complete Graph: Graph where every vertex is connected to every other vertex.
- Bipartite Graph: Graph whose vertices can be divided into two disjoint sets with no edges within each set.
- DAG (Directed Acyclic Graph): Directed graph with no cycles.
- Sparse Graph: Graph with few edges relative to the number of vertices.
- Dense Graph: Graph with many edges relative to the number of vertices.
- Subgraph: Graph formed from a subset of vertices and edges of another graph.
- Bridge: Edge whose removal increases the number of connected components.
- Articulation Point/Cut Vertex: Vertex whose removal increases the number of connected components.
- Minimum Spanning Tree (MST): Subset of edges that form a tree including all vertices with minimum total edge weight.
- Strongly Connected Component: Maximal subgraph where there's a path between any two vertices in both directions.

Interview tips:

- Clarify if the graph is directed or undirected
- Ask about presence of cycles or if it's a DAG (Directed Acyclic Graph)
- Determine if edges are weighted or unweighted
- Consider representation choice based on graph density
- Always track visited nodes to avoid infinite loops in cyclic graphs
- For matrix-based graphs, verify boundary conditions
- Edge cases:
- Empty graph
- Graph with one or two nodes
- Disconnected graphs
- Graph with cycles
- Graphs with negative weight edges (for path-finding)

- Fully connected graphs (complete graphs)
- V is the number of vertices, while E is the number of edges.

Operation	Time	Space	Notes
DFS traversal	O(V + E)	O(V)	Using adjacency list. Space for visited set and recursion stack.
BFS traversal	O(V + E)	O(V)	Using adjacency list. Space for visited set and queue.
Topological sort	O(V + E)	O(V)	Only works on DAGs.
Adding edge	O(1)	O(1)	For adjacency list.
Checking edge	O(1) or O(degree)	O(1)	O(1) for adjacency matrix, O(degree) for adjacency list.
Finding all adjacent	O(1) or O(V)	O(1)	O(V) for adjacency matrix, O(degree) for adjacency list.
Graph construction	O(V + E)	O(V + E)	Converting from edge list to adjacency list/matrix.

Common techniques

Depth-first search

Explores as far as possible along each branch before backtracking. Uses a stack (implicit via recursion or explicit) to track nodes.

When to use: Path finding, cycle detection, topological sorting, connected components.

```
def dfs(matrix):
  # Check for an empty matrix/graph.
  if not matrix:
    return []

  rows, cols = len(matrix), len(matrix[0])
  visited = set()
  directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

  def traverse(i, j):
    if (i, j) in visited:
      return

    visited.add((i, j))
    # Traverse neighbors.
    for direction in directions:
      next_i, next_j = i + direction[0], j + direction[1]
      if 0 <= next_i < rows and 0 <= next_j < cols:
        # Add in question-specific checks, where relevant.
        traverse(next_i, next_j)

  for i in range(rows):
    for j in range(cols):
      traverse(i, j)
```

Problems:

1. https://leetcode.com/problems/pacific-atlantic-water-flow/description/
2. https://leetcode.com/problems/clone-graph/description/

Breadth-first search

Explores all neighbors at the present depth before moving to nodes at the next depth level. Uses a queue to track nodes.

When to use: Shortest path (unweighted), level-order traversal, finding connected components, bipartite graph checking.

from collections import deque

```
def bfs(matrix):
  # Check for an empty matrix/graph.
  if not matrix:
    return []

  rows, cols = len(matrix), len(matrix[0])
  visited = set()
  directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

  def traverse(i, j):
    queue = deque([(i, j)])
    while queue:
      curr_i, curr_j = queue.popleft()
      if (curr_i, curr_j) not in visited:
        visited.add((curr_i, curr_j))
        # Traverse neighbors.
        for direction in directions:
          next_i, next_j = curr_i + direction[0], curr_j + direction[1]
          if 0 <= next_i < rows and 0 <= next_j < cols:
            # Add in question-specific checks, where relevant.
            queue.append((next_i, next_j))

  for i in range(rows):
    for j in range(cols):
      traverse(i, j)
```

Problems:

1. https://leetcode.com/problems/rotting-oranges/
2. https://leetcode.com/problems/pacific-atlantic-water-flow/description/
3. https://leetcode.com/problems/clone-graph/description/

Topological sorting

Produces a linear ordering of vertices such that for every directed edge (u,v), vertex u comes before v in the ordering. Only possible in Directed Acyclic Graphs (DAGs). A topological sort is a graph traversal in which each node v is visited only after all its dependencies are visited.

When to use: Task scheduling, course prerequisites, dependency resolution.

```
from collections import deque
def graph_topo_sort(num_nodes, edges):
    nodes, order, queue = {}, [], deque()
    for node_id in range(num_nodes):
        nodes[node_id] = { 'in': 0, 'out': set() }
    for node_id, pre_id in edges:
        nodes[node_id]['in'] += 1
        nodes[pre_id]['out'].add(node_id)
    for node_id in nodes.keys():
        if nodes[node_id]['in'] == 0:
            queue.append(node_id)
    while len(queue):
        node_id = queue.pop()
        for outgoing_id in nodes[node_id]['out']:
            nodes[outgoing_id]['in'] -= 1
            if nodes[outgoing_id]['in'] == 0:
                queue.append(outgoing_id)
        order.append(node_id)
    return order if len(order) == num_nodes else None

print(graph_topo_sort(4, [[0, 1], [0, 2], [2, 1], [3, 0]]))
# [1, 2, 0, 3]
```

Problems:

1. https://leetcode.com/problems/course-schedule/description/

Matrix

A matrix is a 2-dimensional array of elements arranged in rows and columns. In programming, matrices frequently appear in problems involving spatial relationships, dynamic programming, and graph traversal.

Common data structures and algorithms for matrices:

- Dynamic Programming tables - For problems involving cumulative calculations.
- Graph traversal - When treating each cell as a node connected to adjacent cells.
- Simulation algorithms - For rotating, transforming, or manipulating matrices.
- 2D prefix sums - For quickly calculating sums of rectangular regions.

Interview tips:

- Always clarify the dimensions of the matrix (n × m)
- Ask if the matrix can be modified or needs to be preserved
- Consider boundary conditions (edges and corners) early
- Check if the matrix is square or rectangular
- For graph-like problems, define traversal rules (4-way, 8-way adjacent)
- Think about whether in-place modifications are possible to save space

Edge cases:

- Empty matrix ([] or [[]])
- 1 × 1 matrix (single element)
- Matrix with only one row
- Matrix with only one column
- Matrix with negative values (when relevant)
- Matrix with duplicate values (when relevant)

Operation	Time	Space	Notes
Access element	O(1)	O(1)	Direct indexing with row and column.
Transpose	$O(n^2)$	O(1) or $O(n^2)$	Can be done in-place for square matrices.
Rotate 90°	$O(n^2)$	O(1) or $O(n^2)$	Can be done in-place for square matrices.
Clone/Copy	O(n m)	O(n · m)	Creating a deep copy of the matrix.
2D prefix sum	O(n · m)	O(n · m)	Preprocessing for O(1) range sum queries.
DFS traversal	O(n · m)	O(n · m)	Complete traversal of the matrix.
BFS traversal	O(n · m)	O(n · m)	Level-order traversal of the matrix.

Common techniques

Matrix initialization and copying

Creating new matrices with specific dimensions or copying existing ones.

When to use: Dynamic programming, marking visited states, creating result matrices.

```
# Create an n × m matrix filled with zeros
def create_zero_matrix(n, m):
    return [[0] * m for _ in range(n)]

# Create an n × m matrix filled with a default value
def create_matrix(n, m, default_value):
    return [[default_value] * m for _ in range(n)]

# Deep copy a matrix
def copy_matrix(matrix):
    return [row[:] for row in matrix]
```

Matrix traversal

Visiting all elements in a matrix systematically.

When to use: Grid problems, pathfinding problems, connected components detection.

```
# Depth-First Search (DFS)
def dfs_matrix(matrix, start_row, start_col):
    rows, cols = len(matrix), len(matrix[0])
    visited = [[False] * cols for _ in range(rows)]
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    def dfs(r, c):
        if not (0 <= r < rows and 0 <= c < cols) or visited[r][c]:
            return
        visited[r][c] = True
        for dr, dc in directions:
            dfs(r + dr, c + dc)

    dfs(start_row, start_col)
```

Problems:

1. https://leetcode.com/problems/number-of-islands/

Matrix transformations

Operations that change the structure or arrangement of a matrix.

When to use: Rotations, reflections, transpositions, image processing.

```
# Rotate a matrix 90 degrees clockwise
def rotate_90_clockwise(matrix):
    matrix.reverse()
    for i in range(len(matrix)):
        for j in range(i + 1, len(matrix)):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
```

Problems:

1. https://leetcode.com/problems/rotate-image/

Hash Table

A hash table (also called hash map) is a data structure that implements an associative array abstract data type, mapping keys to values. It uses a hash function to compute an index, or hash code, into an array of buckets or slots, from which the desired value can be found.

Common collision resolution techniques:

- Separate chaining - A linked list is used for each value, so that it stores all the collided items.
- Open addressing - All entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found.

Interview tips:

- Understand that hash tables provide O(1) average time complexity for lookups
- Know when to use hash tables versus other data structures
- Consider the impact of hash collisions on performance
- Remember that dictionaries/objects/maps in most languages are implementations of hash tables

Edge cases:

- Hash collisions (multiple keys hashing to the same index)
- Empty hash table
- High load factor affecting performance
- Poor hash function leading to many collisions
- Handling null or undefined keys

Operation	Time	Space	Notes
Access	N/A	N/A	Accessing not possible as the hash code is not known.
Search	O(1)*	O(1)	* Average case. In interviews we usually only care about average case for hash tables.
Insert	O(1)*	O(1)	* Average case.
Remove	O(1)*	O(1)	* Average case.

Common techniques

Two-sum pattern

Using a hash table to find pairs of elements that sum to a target value.

When to use: Finding complements, pair matching, target sum problems.

```
def two_sum(nums, target):
    seen = {}  # Value -> index
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i
    return None

# Example
nums = [2, 7, 11, 15]
target = 9
print(two_sum(nums, target))  # [0, 1] (2 + 7 = 9)
```

Problems:

1. https://leetcode.com/problems/two-sum/description/

Character Frequency Counting

Using a hash table to count occurrences of characters or elements.

When to use: Anagram detection, character distribution analysis.

```
from collections import Counter

def character_frequency(s):
    # Using Counter
    return Counter(s)

def character_frequency_manual(s):
    # Using dictionary manually
    freq = {}
    for char in s:
        freq[char] = freq.get(char, 0) + 1
    return freq

# Example
s = "programming"
print(character_frequency(s))
# Counter({'r': 2, 'g': 2, 'm': 2, 'p': 1, 'o': 1, 'a': 1, 'i': 1, 'n': 1})
```

Problems:

1. https://leetcode.com/problems/group-anagrams/description/

Recursion

Recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem. Many algorithms relevant in coding interviews make heavy use of recursion - binary search, merge sort, tree traversal, depth-first search, etc.

Key components:

- Base case(s): Define when the recursion stops
- Recursive case: Breaking down the problem into smaller subproblems

Interview tips:

- Always remember to define a base case so your recursion will end
- Recursion is useful for permutation problems and tree-based questions
- Recursion implicitly uses a stack, which can lead to stack overflow for deep recursion
- Number of base cases depends on the recursive calls - for fibonacci, you need two base cases since it calls `fib(n-1)` and `fib(n-2)`

Edge cases:

- n = 0
- n = 1
- Ensure all base cases are covered

Operation	Time	Space	Notes
Simple recursion	$O(branches^{depth})$	O(depth)	Space used by the recursion stack
Tail recursion	$O(branches^{depth})$	O(1)*	* Only with proper tail call optimization
Memoized recursion	O(states)	O(states)	Additional space used for memoization cache

Common techniques

Basic recursion

Define base case(s) and recursive case.

When to use: Tree traversal, divide and conquer algorithms, problems with recursive structure.

```
def factorial(n):
    # Base case
    if n <= 1:
        return 1

    # Recursive case
    return n * factorial(n - 1)

# Example: Fibonacci sequence
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Memoization

Cache results of function calls to avoid redundant computation.

When to use: When recursive function makes multiple calls with the same arguments (overlapping subproblems).

```
# Fibonacci with memoization
def fibonacci_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n

    memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)
    return memo[n]

# Example with explicit memo dictionary
def fib_memo(n):
    memo = {}

    def dp(i):
        if i in memo:
            return memo[i]
        if i <= 1:
            return i

        memo[i] = dp(i - 1) + dp(i - 2)
        return memo[i]

    return dp(n)
```

Problems:

1. https://leetcode.com/problems/generate-parentheses/
2. https://leetcode.com/problems/combinations/
3. https://leetcode.com/problems/subsets/
4. https://leetcode.com/problems/letter-combinations-of-a-phone-number/

Linked List

Linked lists are used to represent sequential data. It's a linear collection of data elements whose order is not given by their physical placement in memory, as opposed to arrays, where data is stored in sequential blocks of memory.

Advantages:

- O(1) insertion and deletion at known position
- Dynamic size
- No need for contiguous memory allocation

Disadvantages:

- O(n) access time (can't directly index)
- Extra memory for pointers
- Not cache-friendly due to non-contiguous memory

Types of linked lists:

- Singly linked list: Each node points to the next node, last node points to null
- Doubly linked list: Each node has two pointers, next and prev
- Circular linked list: Last node points back to first node

Interview tips:

- Consider using dummy/sentinel nodes to simplify edge cases
- Two-pointer technique is very common (slow/fast pointers)
- Always check for null pointers
- Consider whether in-place modification is required

Edge cases:

- Empty linked list (head is null)
- Single node
- Two nodes
- Linked list has cycles

Operation	Time	Space	Notes
Access	O(n)	O(1)	Must traverse from head
Search	O(n)	O(1)	Linear traverse
Insert	O(1)	O(1)	Once position is found
Remove	O(1)	O(1)	Once position is found
Traverse	O(n)	O(1)	Visit all nodes

Common techniques

Two pointers

Use two pointers moving at different speeds or from different positions.

When to use: Finding middle element, detecting cycles, getting kth element from end, etc.

```
# Find middle of linked list
def find_middle(head):
    if not head or not head.next:
        return head

    slow = fast = head

    # Fast moves twice as fast as slow
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    return slow

# Detect cycle
def has_cycle(head):
    if not head or not head.next:
        return False

    slow = fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            return True

    return False
```

Reversing a linked list

Change the direction of pointers to reverse the list.

When to use: When the order needs to be inverted, or as a technique for other list problems.

```
def reverse_list(head):
    prev = None
    current = head

    while current:
        next_temp = current.next  # Store next node
        current.next = prev       # Reverse pointer
        prev = current            # Move prev forward
        current = next_temp       # Move current forward

    return prev  # New head is the previous tail
```

Problems:

1. https://leetcode.com/problems/reverse-linked-list/
2. https://leetcode.com/problems/linked-list-cycle/
3. https://leetcode.com/problems/merge-two-sorted-lists/
4. https://leetcode.com/problems/remove-nth-node-from-end-of-list/

Queue

Queue is an abstract data type that maintains a linear collection of elements following FIFO (First In, First Out) principle. Elements are added at one end (rear/tail) and removed from the other end (front/head).

Common implementations:

- Array-based queues
- Linked list-based queues
- Circular queues

Interview tips:

- Remember that queue implementations using arrays require tracking front and rear indices
- Be familiar with both array and linked list implementations
- Know that BFS commonly uses queues
- When implementing queues with lists in Python, dequeue will be O(n) due to shifting

Edge cases:

- Empty queue
- Queue with one item
- Queue with two items
- Full queue (for fixed-size implementations)

Operation	Time	Space	Notes
Enqueue/Offer	O(1)	O(1)	Add to back of queue
Dequeue/Poll	O(1)	O(1)	Remove from front of queue
Front/Peek	O(1)	O(1)	View front element without removing
IsEmpty	O(1)	O(1)	Check if queue is empty
Size	O(1)	O(1)	Get number of elements

Common techniques

Simple queue implementation

Basic queue operations using a linked list or dynamic array.

When to use: When FIFO order is required for processing elements.

```
from collections import deque

class Queue:
    def __init__(self):
        self.items = deque()

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.popleft()
        return None

    def front(self):
        if not self.is_empty():
            return self.items[0]
        return None

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)
```

Queue using stacks

Implement a queue using two stacks.

When to use: When only stack operations are available.

```
class QueueUsingStacks:
    def __init__(self):
        self.stack1 = []  # For enqueue
        self.stack2 = []  # For dequeue

    def enqueue(self, item):
        self.stack1.append(item)

    def dequeue(self):
        # If stack2 is empty, transfer all elements from stack1
        if not self.stack2:
            while self.stack1:
                self.stack2.append(self.stack1.pop())

        if self.stack2:
            return self.stack2.pop()
        return None

    def front(self):
        if not self.stack2:
            while self.stack1:
                self.stack2.append(self.stack1.pop())

        if self.stack2:
            return self.stack2[-1]
        return None

    def is_empty(self):
        return len(self.stack1) == 0 and len(self.stack2) == 0
```

Problems:

1. https://leetcode.com/problems/implement-stack-using-queues
2. https://leetcode.com/problems/implement-queue-using-stacks
3. https://leetcode.com/problems/design-circular-queue

Stack

Stack is an abstract data type that supports operations push (insert on top) and pop (remove from top), following the LIFO (Last In, First Out) principle.

Common implementations:

- Array-based stacks
- Linked list-based stacks

Interview tips:

- Remember that stacks are used for DFS, backtracking, expression evaluation
- Consider problems involving matching parentheses or brackets
- Watch for stack overflow in recursive implementations
- Useful for problems requiring tracking of operations that need to be undone

Edge cases:

- Empty stack (popping from empty stack)
- Stack with one item
- Stack with two items
- Full stack (for fixed-size implementations)

Operation	Time	Space	Notes
Push	O(1)	O(1)	Add element to top
Pop	O(1)	O(1)	Remove element from top
Top/Peek	O(1)	O(1)	View top element without removing
IsEmpty	O(1)	O(1)	Check if stack is empty
Search	O(n)	O(1)	Find element in stack

Common techniques

Simple stack implementation

Basic stack operations using an array or linked list.

When to use: When LIFO order is required, expression evaluation, backtracking.

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return None

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return None

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)
```

Parentheses matching

Use stack to check if parentheses are balanced.

When to use: Validation of expressions, matching brackets/parentheses.

```
def is_valid_parentheses(s):
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}

    for char in s:
        if char in mapping:  # closing bracket
            top_element = stack.pop() if stack else '#'
            if mapping[char] != top_element:
                return False
        else:  # opening bracket
            stack.append(char)

    return not stack  # stack should be empty at the end
```

Problems:

1. https://leetcode.com/problems/valid-parentheses
2. https://leetcode.com/problems/min-stack
3. https://leetcode.com/problems/evaluate-reverse-polish-notation
4. https://leetcode.com/problems/daily-temperatures

Interval

Interval questions are a subset of array questions where you are given an array of two-element arrays (an interval) and the two values represent a start and an end value.

Common operations:

- Merging overlapping intervals
- Finding intersections of intervals
- Inserting new intervals

Interview tips:

- Clarify whether [1, 2] and [2, 3] are considered overlapping
- Confirm whether intervals will strictly follow a < b
- Sorting by start or end point is often a key first step
- Consider line sweep algorithms for more complex interval problems

Edge cases:

- No intervals
- Single interval
- Two intervals
- Non-overlapping intervals
- An interval totally consumed within another interval
- Duplicate intervals (exactly the same start and end)
- Intervals which start right where another interval ends ([1, 2], [2, 3])

Operation	Time	Space	Notes
Sort intervals	O(n · log(n))	O(1) or O(n)	By start or end point
Merge intervals	O(n · log(n))	O(n)	Including sort time
Insert interval	O(n)	O(n)	If already sorted
Check overlap	O(1)	O(1)	For two intervals

Common techniques

Merge overlapping intervals

Sort intervals by start time and merge overlapping ones.

When to use: Combining overlapping time ranges, simplifying interval representations.

```
def merge_intervals(intervals):
    if not intervals:
        return []

    # Sort by start time
    intervals.sort(key=lambda x: x[0])

    merged = [intervals[0]]

    for current in intervals[1:]:
        # Get the last added interval
        previous = merged[-1]

        # If current interval overlaps with previous
        if current[0] <= previous[1]:
            # Merge them by updating end time
            previous[1] = max(previous[1], current[1])
        else:
            # Add as a new interval
            merged.append(current)

    return merged
```

Check if intervals overlap

Determine if two intervals have any overlap.

When to use: Collision detection, scheduling conflicts.

```
def is_overlap(a, b):
    return a[0] < b[1] and b[0] < a[1]

# Merge two overlapping intervals
def merge_overlapping_intervals(a, b):
    return [min(a[0], b[0]), max(a[1], b[1])]
```

Problems:

1. https://leetcode.com/problems/merge-intervals/
2. https://leetcode.com/problems/insert-interval/
3. https://leetcode.com/problems/non-overlapping-intervals/

Heap

Heaps are specialized tree-based data structures that satisfy the heap property. A heap is a complete binary tree where each node is either greater than or equal to (max heap) or less than or equal to (min heap) its children.

Types of heaps:

- Max Heap: parent nodes are greater than or equal to their children
- Min Heap: parent nodes are less than or equal to their children

Interview tips:

- If you see a "top k" or "kth largest/smallest" in the problem, consider using a heap
- For top k largest elements, use a min heap of size k
- For top k smallest elements, use a max heap of size k
- In Python, **heapq** implements a min heap by default

Edge cases:

- Empty heap
- Heap with duplicate values
- Heap with negative values

Operation	Time	Space	Notes
Find min/max	O(1)	O(1)	Peek at root
Insert	O(log(n))	O(1)	Add and bubble up
Remove min/max	O(log(n))	O(1)	Remove root and bubble down
Heapify	O(n)	O(1)	Build heap from array

Common techniques

Top K elements

Use a heap to efficiently find the k largest or smallest elements.

When to use: Finding top k elements, kth largest/smallest number.

```
import heapq

# Find k largest elements
def find_k_largest(nums, k):
    # Use a min heap of size k
    min_heap = []

    for num in nums:
        # Push elements onto heap
        heapq.heappush(min_heap, num)

        # Keep heap size <= k
        if len(min_heap) > k:
            heapq.heappop(min_heap)

    # Result will be in reverse order
    return sorted(min_heap, reverse=True)

# Example
print(find_k_largest([3, 1, 5, 12, 2, 11], 3))  # [5, 11, 12]

# Using heapq.nlargest (more concise)
def find_k_largest_simple(nums, k):
    return heapq.nlargest(k, nums)
```

Merge K sorted lists/arrays

Use a min heap to efficiently merge multiple sorted lists.

When to use: Merging k sorted lists, merging data from multiple sources.

```
import heapq

def merge_k_sorted_lists(lists):
    result = []
    min_heap = []

    # Initialize heap with first element from each list
    for i, lst in enumerate(lists):
        if lst:  # Ensure the list is not empty
            heapq.heappush(min_heap, (lst[0], i, 0))  # (value, list_index, element_index)

    # Process elements
    while min_heap:
        val, list_idx, elem_idx = heapq.heappop(min_heap)
        result.append(val)

        # If there are more elements in this list
        if elem_idx + 1 < len(lists[list_idx]):
            next_elem_idx = elem_idx + 1
            heapq.heappush(min_heap, (lists[list_idx][next_elem_idx], list_idx, next_elem_idx))

    return result
```

Problems:

1. https://leetcode.com/problems/merge-k-sorted-lists/
2. https://leetcode.com/problems/k-closest-points-to-origin/
3. https://leetcode.com/problems/top-k-frequent-elements/
4. https://leetcode.com/problems/find-median-from-data-stream/

Trie

Tries (prefix trees) are tree-like data structures used to store a dynamic set of strings, where keys are usually strings. Tries are particularly efficient for dictionary operations like lookup, insert, and delete.

Characteristics:

- Each node can have multiple children, one for each character
- The root represents an empty string
- Each path from root to a marked node represents a word

Interview tips:

- Be familiar with implementing a trie from scratch
- Know the methods: add, search, startsWith
- Consider tries for autocomplete or spelling checker problems
- Tries can be more efficient than hash tables for prefix-based operations

Edge cases:

- Empty string
- Searching for a string in an empty trie
- Inserting duplicate strings

Operation	Time	Space	Notes
Insert	O(m)	O(m)	m = length of word
Search	O(m)	O(1)	m = length of word
Starts with	O(m)	O(1)	m = length of prefix
Delete	O(m)	O(1)	m = length of word

Common techniques

Implement basic trie

Create a trie with insert, search, and startsWith methods.

When to use: Dictionary operations, prefix searches, autocomplete.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

    def starts_with(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True
```

Word dictionary with wildcards

Extend the trie to handle wildcard characters in searches.

When to use: Pattern matching, word games, searches with unknown characters.

```
def search_with_wildcard(self, word):
    def dfs(node, index):
        if index == len(word):
            return node.is_end_of_word

        if word[index] == '.':
            # Try all possible characters
            for child in node.children.values():
                if dfs(child, index + 1):
                    return True
            return False
        else:
            # Normal character
            if word[index] not in node.children:
                return False
            return dfs(node.children[word[index]], index + 1)

    return dfs(self.root, 0)
```

Problems:

1. https://leetcode.com/problems/implement-trie-prefix-tree
2. https://leetcode.com/problems/add-and-search-word-data-structure-design
3. https://leetcode.com/problems/word-search-ii/

Dynamic Programming

Dynamic Programming (DP) is an algorithmic paradigm that solves complex problems by breaking them down into simpler subproblems and storing the results to avoid redundant computations.

Key characteristics:

- Overlapping subproblems: Same subproblems solved multiple times
- Optimal substructure: Optimal solution can be constructed from optimal solutions of subproblems

Implementation approaches:

- Top-down (memoization): Recursive approach with caching
- Bottom-up (tabulation): Iterative approach building solutions from smallest subproblems

Interview tips:

- Identify if a problem can be solved with DP (look for overlapping subproblems)
- Define the state clearly (what does each entry in your DP table represent)
- Establish the recurrence relation (how states relate to each other)
- Consider space optimization (sometimes you only need the last few states)
- Be careful with initialization values

Edge cases:

- Empty input
- Single element
- No solution exists
- Multiple solutions with same optimality

Approach	Time	Space	Notes
Brute Force	O(exponential)	O(n)	Recursive without memoization
Top-down DP	O(states · work per state)	O(states)	With memoization
Bottom-up DP	O(states · work per state)	O(states)	Iterative with tabulation

Common techniques

1D Dynamic Programming

Problems where state depends on previous computed values.

When to use: Subsequence problems, optimization problems with 1D input.

```
# Fibonacci sequence using bottom-up DP
def fibonacci(n):
    if n <= 1:
        return n

    dp = [0] * (n + 1)
    dp[0], dp[1] = 0, 1

    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]

# Space-optimized version
def fibonacci_optimized(n):
    if n <= 1:
        return n

    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b

    return b
```

2D Dynamic Programming

Problems where state depends on results from multiple dimensions.

When to use: Grid problems, string comparison, interval problems.

```
# Longest Common Subsequence
def longest_common_subsequence(text1, text2):
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]
```

Problems:

1. https://leetcode.com/problems/climbing-stairs/
2. https://leetcode.com/problems/coin-change/
3. https://leetcode.com/problems/longest-increasing-subsequence/
4. https://leetcode.com/problems/longest-common-subsequence/
5. https://leetcode.com/problems/unique-paths/

Binary

Binary operations involve manipulating individual bits in numbers. Bit manipulation is often used in low-level programming and can provide efficient solutions for certain problems.

Common bit operations:

- AND (&): Sets bits that are 1 in both operands
- OR (|): Sets bits that are 1 in either operand
- XOR (^): Sets bits that are different between operands
- NOT (~): Flips all bits
- Left shift (<<): Shifts bits left, multiplying by 2^k
- Right shift (>>): Shifts bits right, dividing by 2^k

Interview tips:

- Know how to convert between decimal and binary
- Be careful with handling negative numbers (two's complement)
- Understand bitwise operators in your language
- Watch for integer overflow/underflow

Edge cases:

- Zero values
- Negative numbers
- Overflow/underflow
- All bits set or all bits clear

Operation	Time	Space	Notes
Bitwise operations	O(1)	O(1)	AND, OR, XOR, NOT
Bit shifting	O(1)	O(1)	Left/right shifts
Counting bits	O(log(n))	O(1)	Number of set bits

Common techniques

Check/set/clear specific bits

Manipulate individual bits at specific positions.

When to use: Flag operations, state management using bits, optimization.

```
# Test if kth bit is set
def is_kth_bit_set(num, k):
    return (num & (1 << k)) != 0

# Set kth bit
def set_kth_bit(num, k):
    return num | (1 << k)

# Clear kth bit
def clear_kth_bit(num, k):
    return num & ~(1 << k)

# Toggle kth bit
def toggle_kth_bit(num, k):
    return num ^ (1 << k)
```

Bit counting and manipulation

Count set bits and perform common bit operations.

When to use: Optimization problems, state compression, number theory.

```
# Count number of set bits (Brian Kernighan's algorithm)
def count_set_bits(num):
    count = 0
    while num:
        num &= (num - 1)  # Clear the least significant set bit
        count += 1
    return count

# Check if number is power of 2
def is_power_of_two(num):
    return num > 0 and (num & (num - 1)) == 0

# Get the rightmost set bit
def get_rightmost_set_bit(num):
    return num & -num
```

Problems:

1. https://leetcode.com/problems/sum-of-two-integers/
2. https://leetcode.com/problems/number-of-1-bits/
3. https://leetcode.com/problems/counting-bits/
4. https://leetcode.com/problems/single-number/

Math

Mathematical algorithms and concepts are essential in many programming problems. Understanding these can provide elegant solutions to seemingly complex problems.

Common mathematical concepts:

- Number theory (primes, GCD, LCM)
- Combinatorics (permutations, combinations)
- Probability and statistics

Modular arithmetic

Operation	Formula	Time complexity
Check if a number is even	num % 2 == 0	O(1) Constant time, single operation
Sum of 1 to N	1 + 2 + ... + (N - 1) + N = (N · (N+1)) / 2	O(1) Direct formula, no iteration needed
Sum of Geometric Progression	2^0 + 2^1 + 2^2 + 2^3 + ... + 2^n = 2^{n+1} - 1	O(1) Direct formula, computed in constant time
Permutations of N, picking K numbers	N! / (N-K)!	O(N!) worst case
Combinations of N, picking K numbers	N! / (K! · (N-K)!)	O(N!) worst case, but often O(2^N) or O(N^K) depending on the computation method

Interview tips:

- Check for division by zero
- Handle overflow/underflow in typed languages
- Consider edge cases like negative numbers and floating point precision
- Look for mathematical patterns that simplify the problem

Edge cases:

- Division by 0
- Multiplication by 1
- Negative numbers
- Very large numbers (overflow)
- Floating point precision issues

Operation	Time	Space	Notes
Basic arithmetic	O(1)	O(1)	Addition, subtraction, etc.
Exponentiation	O(log(n))	O(1)	Using fast power algorithm
GCD/LCM	O(log(min(a,b)))	O(1)	Using Euclidean algorithm
Prime factorization	O(√n)	O(log(n))	Number of prime factors

Common techniques

Fast exponentiation

Calculate power efficiently using divide and conquer.

When to use: Computing large powers, modular exponentiation.

```
# Calculate x^n efficiently
def fast_power(x, n):
    if n < 0:
        return 1 / fast_power(x, -n)
    if n == 0:
        return 1

    # Calculate half power recursively
    half = fast_power(x, n // 2)

    # If n is even: x^n = (x^(n/2))^2
    if n % 2 == 0:
        return half * half
    # If n is odd: x^n = x * (x^(n/2))^2
    else:
        return x * half * half
```

GCD and LCM

Calculate greatest common divisor and least common multiple.

When to use: Fraction simplification, problems involving divisibility.

```
# Euclidean algorithm for GCD
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

# LCM using GCD
def lcm(a, b):
    return a * b // gcd(a, b)
```

Problems:

1. https://leetcode.com/problems/powx-n/
2. https://leetcode.com/problems/sqrtx/
3. https://leetcode.com/problems/integer-to-english-words/

Geometry

Geometry problems involve points, lines, shapes, and their properties. While less common in typical coding interviews, some companies may ask these questions.

Common geometric entities:

- Point (x, y coordinates)
- Line (slope, intercept)
- Rectangle (corners, width, height)
- Circle (center, radius)
- Polygon (vertices)

Interview tips:

- Be careful with floating-point comparisons (use epsilon comparisons)
- Consider edge cases like parallel lines, collinear points
- Know how to check if points are inside shapes
- For distance calculations, using squared distance is often sufficient

Edge cases:

- Zero values (radius, length, etc.)
- Parallel lines
- Collinear points
- Degenerate shapes (zero area)
- Overlapping shapes

Operation	Time	Space	Notes
Distance between points	O(1)	O(1)	Use squared distance to avoid square root
Check if point in shape	O(1)	O(1)	For simple shapes like circles or rectangles
Check if shapes overlap	O(1)	O(1)	Rectangles, circles
Convex hull	O(n · log(n))	O(n)	Graham scan or other algorithms

Common techniques

Rectangle overlap check

Determine if two rectangles overlap.

When to use: Collision detection, layout analysis.

```
def is_rectangle_overlap(rect1, rect2):
    # rect format: [x1, y1, x2, y2] where (x1,y1) is bottom-left, (x2,y2) is top-right
    return (rect1[0] < rect2[2] and  # left edge of rect1 < right edge of rect2
            rect1[2] > rect2[0] and  # right edge of rect1 > left edge of rect2
            rect1[1] < rect2[3] and  # bottom edge of rect1 < top edge of rect2
            rect1[3] > rect2[1])     # top edge of rect1 > bottom edge of rect2
```

Circle overlap check

Determine if two circles overlap.

When to use: Collision detection, range overlap.

```
def is_circle_overlap(c1, c2, r1, r2):
    # c1, c2 are centers (x,y), r1, r2 are radii
    dx = c1[0] - c2[0]
    dy = c1[1] - c2[1]
    distance_squared = dx * dx + dy * dy

    # Compare with square of sum of radii
    return distance_squared <= (r1 + r2) * (r1 + r2)
```

Problems:

1. https://leetcode.com/problems/rectangle-overlap/
2. https://leetcode.com/problems/k-closest-points-to-origin/
3. https://leetcode.com/problems/rectangle-area/