The Little Book of Algorithms

Version 0.3.0

Duc-Tam Nguyen

2025-10-06

Table of contents

Content

A Friendly Guide from Numbers to Neural Networks

- Download PDF - print-ready
- Download EPUB - e-reader friendly
- View LaTex - `.tex` source
- Source code (Github) - Markdown source
- Read on GitHub Pages - view online

The Book

- Download PDF - print-ready
- Download EPUB - e-reader friendly
- View LaTex - `.tex` source
- Source code (Github) - Markdown source
- Read on GitHub Pages - view online

Chapter 1. Foundations of algorithms

1. What Is an Algorithm?

Let's begin at the very heart of computer science. Before we dive into code, data, or performance, we need to understand what an algorithm *really is*.

An algorithm is a clear, step-by-step procedure to solve a problem. Think of it like a recipe: you have inputs (ingredients), a series of steps (instructions), and an output (the finished dish).

At its core, an algorithm must be:

- Precise: Every step is well-defined and unambiguous.- Finite: It must finish after a certain number of steps.- Effective: Each step is doable by a machine or human.- Deterministic (usually): Same input, same output. When you write an algorithm, you're describing *how* to get from question to answer, not just *what* the answer is.

Example: A Simple Sum

Suppose you want to find the sum of numbers from 1 to (n).

You could describe it like this:

1. Start with total = 0
2. For i = 1 to n 3. Add i to total
3. Return total

That's an algorithm! It's clear, finite, and mechanical.

You can express it in pseudocode:

```
Algorithm SumToN(n):
    total ← 0
    for i ← 1 to n:
        total ← total + i
    return total
```

Or even write it in C:

```c
int sum_to_n(int n) {
    int total = 0;
    for (int i = 1; i <= n; i++) {
        total += i;
    }
    return total;
}
```

Tiny Code

Let's run a quick example by hand.

If (n = 5):

- total = 0- Add 1 → total = 1- Add 2 → total = 3- Add 3 → total = 6- Add 4 → total = 10- Add 5 → total = 15 Output = 15

We can even derive a formula later (you'll see it soon!):

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Why It Matters

Algorithms are the blueprints of computation. Every software system, from a calculator to an AI model, is built from algorithms.

Without algorithms, computers would be powerless , they're just fast machines following instructions. Algorithms give them purpose and direction.

Understanding algorithms means you understand how thinking turns into code.

Later, you'll learn to analyze them (how fast, how big, how smart), but for now, remember:

Algorithms are the language of problem solving.

Try It Yourself

1. Write a simple algorithm to find the maximum number in a list.
2. Write an algorithm to reverse a string.
3. For extra fun, describe your morning routine as an algorithm (inputs, steps, output).

The best way to learn is to think algorithmically , break problems into small, clear, executable steps.

2. Measuring Time and Space

Now that you know what an algorithm is, it's time to ask a deeper question:

How do we know if one algorithm is *better* than another?

It's not enough for an algorithm to be correct , it should also be *efficient*. We measure efficiency in two key ways: time and space.

Time Complexity

Time means how long an algorithm takes to run, relative to its input size. We don't measure in seconds, because hardware changes. Instead, we count steps or operations.

For example:

```
for (int i = 0; i < n; i++) {
    printf("Hi\n");
}
```

This loop runs (n) times, so we say it has time complexity O(n) , it grows linearly with input size.

Another example:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    printf("*");
```

This one runs $n \times n = n^2$ times $\rightarrow$ O(n^2).

You'll see these "Big-O" symbols a lot , they express how runtime grows as inputs grow.

Space Complexity

Space means how much memory the algorithm uses.

Example:

```
int sum = 0;         // O(1) space
```

Uses a constant amount of memory.

But an array like:

```
int arr[n];          // O(n) space
```

uses space proportional to the input size.

Sometimes you'll trade time for space. For example:

- Using a hash table speeds up lookups (more memory, less time)- Using a streaming algorithm saves memory (less space, more time)

Tiny Code

Let's compare two ways to compute a sum from 1 to n:

Method 1: Loop

```
int sum_loop(int n) {
    int total = 0;
    for (int i = 1; i <= n; i++) total += i;
    return total;
}
```

- Time: (O(n))- Space: (O(1)) Method 2: Formula

```
int sum_formula(int n) {
    return n * (n + 1) / 2;
}
```

- Time: (O(1))- Space: (O(1)) Both are correct , but one is *faster*. That's why analyzing time and space matters.

Why It Matters

When data grows huge (millions, billions), small inefficiencies explode.

An algorithm that takes (On^2) might seem fine for 10 elements, but impossible for 1,000,000.

Knowing how to measure time and space helps you:

- Predict performance- Compare solutions- Optimize intelligently It's your compass for navigating complexity.

Try It Yourself

1. Write a simple algorithm to find the minimum in an array. Estimate its time and space complexity.
2. Compare two algorithms that solve the same problem , which one scales better?
3. Can you think of a task in daily life that's O(n)? What about O(1)?

Understanding these measurements early will make every algorithm you meet more meaningful.

3. Big-O, Big-Theta, Big-Omega

Now that you can *measure* time and space, let's learn the language used to describe those measurements.

When we say an algorithm is O(n), we're using asymptotic notation , a way to describe how an algorithm's running time or memory grows as input size (n) increases.

It's not about exact steps, but how the cost *scales* for very large (n).

The Big-O (Upper Bound)

Big-O answers: *"How bad can it get?"* It gives an upper bound on growth , the worst-case scenario.

If an algorithm takes at most (5n + 20) steps, we write (O(n)). We drop constants and lower-order terms because they don't matter at scale.

Common Big-O notations:

Name	Notation	Growth	Example
Constant	O(1)	Flat	Accessing array element
Logarithmic	O(log n)	Very slow growth	Binary search

Name	Notation	Growth	Example
Linear	O(n)	Proportional	Single loop
Quadratic	O(n^2)	Grows quickly	Double loop
Exponential	O(2)	Explodes	Recursive subset generation

So if your algorithm is (O(n)), doubling input size doubles runtime. If it's (On^2), doubling input size makes it four times slower.

The Big-Theta (Tight Bound)

Big-Theta (Θ) gives a tight bound , when you know the algorithm's growth from above and below.

If runtime is roughly (3n + 2), then (T(n) = Θ(n)). That means it's both (O(n)) and (Ω(n)).

The Big-Omega (Lower Bound)

Big-Omega (Ω) answers: *"How fast can it possibly be?"* It's the best-case growth , the lower limit.

Example:

- Linear search: (Ω(1)) if the element is at the start- (O(n)) in the worst case if it's at the end So we might say:
$$T(n) = \Omega(1),\ \Theta(n) = O(n)$$

Tiny Code

Let's see Big-O in action.

```c
int sum_pairs(int n) {
    int total = 0;
    for (int i = 0; i < n; i++)         // O(n)
        for (int j = 0; j < n; j++)     // O(n)
            total += i + j;             // O(1)
    return total;
}
```

Total steps $n \times n = n^2$. So (T(n) = On^2).

If we added a constant-time operation before or after the loops, it wouldn't matter. Constants vanish in asymptotic notation.

Why It Matters

Big-O, Big-Theta, and Big-Omega let you talk precisely about performance. They're like the grammar of efficiency.

When you can write:

> "Algorithm A runs in O(n log n) time, O(n) space," you've captured its essence clearly and compared it meaningfully.

They help you:

- Predict behavior at scale- Choose better data structures- Communicate efficiency in interviews and papers It's not about exact timing , it's about growth.

Try It Yourself

1. Analyze this code:

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

   What's the time complexity?
2. Write an algorithm that's O(n log n) , hint: merge sort!
3. Identify the best, worst, and average-case complexities for linear search and binary search.

Learning Big-O is like learning a new language , once you're fluent, you can *see* how code grows before you even run it.

4. Algorithmic Paradigms (Greedy, Divide and Conquer, DP)

Once you can measure performance, it's time to explore how algorithms are designed. Behind every clever solution is a guiding paradigm , a way of thinking about problems.

Three of the most powerful are:

1. Greedy Algorithms
2. Divide and Conquer

3. Dynamic Programming (DP)

Each one represents a different mindset for problem-solving.

1. Greedy Algorithms

A greedy algorithm makes the best local choice at each step, hoping it leads to a global optimum.

Think of it like:

> "Take what looks best right now, don't worry about the future."

They're fast and simple, but not always correct , they only work when the greedy choice property holds.

Example: Coin Change (Greedy version) Suppose you want to make 63 cents using US coins (25, 10, 5, 1). The greedy approach:

- Take 25 $\rightarrow$ 38 left- Take 25 $\rightarrow$ 13 left- Take 10 $\rightarrow$ 3 left- Take 1 $\times$ 3 Works here, but not always (try coins 1, 3, 4 for 6).

Simple but not guaranteed optimal.

Common greedy algorithms:

- Kruskal's MST- Prim's MST- Dijkstra's (non-negative weights)- Huffman coding

2. Divide and Conquer

This one's a classic. You break the problem into smaller subproblems, solve each recursively, and then combine the results.

It's like splitting a task among friends, then merging the answers.

Formally:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Examples:

- Merge Sort: divide the array, sort halves, merge- Quick Sort: partition around pivot- Binary Search: halve the range each step Elegant and powerful, but recursion overhead can add cost if poorly structured.

3. Dynamic Programming (DP)

DP is for problems with overlapping subproblems and optimal substructure. You solve smaller subproblems once and store the results to avoid recomputation.

It's like divide and conquer with memory.

Example: Fibonacci Naive recursion: exponential DP with memoization: linear

```c
int fib(int n) {
    if (n <= 1) return n;
    static int memo[1000] = {0};
    if (memo[n]) return memo[n];
    memo[n] = fib(n-1) + fib(n-2);
    return memo[n];
}
```

Efficient reuse, but requires insight into subproblem structure.

Tiny Code

Here's a quick comparison using Fibonacci:

Naive (Divide and Conquer)

```c
int fib_dc(int n) {
    if (n <= 1) return n;
    return fib_dc(n-1) + fib_dc(n-2);  // exponential
}
```

DP (Memoization)

```c
int fib_dp(int n, int memo[]) {
    if (n <= 1) return n;
    if (memo[n]) return memo[n];
    return memo[n] = fib_dp(n-1, memo) + fib_dp(n-2, memo);
}
```

Why It Matters

Algorithmic paradigms give you patterns for design:

- Greedy: when local choices lead to global optimum- Divide & Conquer: when the problem splits naturally- Dynamic Programming: when subproblems overlap Once you recognize a problem's structure, you'll instantly know which mindset fits best.

Think of paradigms as *templates for reasoning* , not just techniques, but philosophies.

Try It Yourself

1. Write a greedy algorithm to make change using coins [1, 3, 4] for amount 6. Does it work?
2. Implement merge sort using divide and conquer.
3. Solve Fibonacci both ways (naive vs DP) and compare speeds.
4. Can you think of a real-life task you solve greedily?

Learning paradigms is like learning styles of thought , once you know them, every problem starts to look familiar.

5. Recurrence Relations

Every time you break a problem into smaller subproblems, you create a recurrence , a mathematical way to describe how the total cost grows.

Recurrence relations are the backbone of analyzing recursive algorithms. They tell us how much time or space an algorithm uses, based on the cost of its subproblems.

What Is a Recurrence?

A recurrence relation expresses (T(n)), the total cost for input size (n), in terms of smaller instances.

Example: For merge sort:

$$T(n) = 2T(n/2) + O(n)$$

That means:

- It divides the problem into 2 halves (2T(n/2))- Merges results in O(n) time You'll often see recurrences like:
- (T(n) = T(n - 1) + O(1))- (T(n) = 2T(n/2) + O(n))- (T(n) = T(n/2) + O(1)) Each one represents a different structure of recursion.

Example 1: Simple Linear Recurrence

Consider this code:

```c
int count_down(int n) {
    if (n == 0) return 0;
    return 1 + count_down(n - 1);
}
```

This calls itself once for each smaller input →

$$T(n) = T(n-1) + O(1)$$

Solve it:

$$T(n) = O(n)$$

Because it runs once per level.

Example 2: Binary Recurrence

For binary recursion:

```c
int sum_tree(int n) {
    if (n == 1) return 1;
    return sum_tree(n/2) + sum_tree(n/2) + 1;
}
```

Here we do two subcalls on (n/2) and a constant amount of extra work.

$$T(n) = 2T(n/2) + O(1)$$

Solve it: (T(n) = O(n))

Why? Each level doubles calls, but halves size. There are log n levels, and total work adds up to O(n).

Solving Recurrences

There are several ways to solve them:

1. Substitution Method Guess the solution, then prove it by induction.
2. Recursion Tree Method Expand the recurrence into a tree and sum the cost per level.
3. Master Theorem Use a formula when recurrence matches:

$$T(n) = aT(n/b) + f(n)$$

Master Theorem (Quick Summary)

If (T(n) = aT(n/b) + f(n)), then:

- If (f(n) = O$n^{\log_b a - \epsilon}$), then (T(n) = Θ$n^{\log_b a}$)- If (f(n) = Θ$n^{\log_b a}$), then (T(n) = Θ$n^{\log_b a} \log n$)- If (f(n) = Ω$n^{\log_b a + \epsilon}$), and regularity condition holds, then (T(n) = Θ(f(n))) Example: Merge Sort → (a = 2, b = 2, f(n) = O(n))

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

Tiny Code

Let's write a quick recursive sum:

```c
int sum_array(int arr[], int l, int r) {
    if (l == r) return arr[l];
    int mid = (l + r) / 2;
    return sum_array(arr, l, mid) + sum_array(arr, mid+1, r);
}
```

Recurrence:

$$T(n) = 2T(n/2) + O(1)$$

→ (O(n))

If you added merging (like in merge sort), you'd get (+O(n)), → (O$n \log n$)

Why It Matters

Recurrence relations let you predict the cost of recursive solutions.

Without them, recursion feels like magic , with them, you can quantify efficiency.

They're key to understanding:

- Divide and conquer- Dynamic programming- Backtracking Once you can set up a recurrence, solving it becomes a game of algebra and logic.

Try It Yourself

1. Write a recurrence for binary search. Solve it.
2. Write a recurrence for merge sort. Solve it.
3. Analyze this function:

   ```
   void fun(int n) {
       if (n <= 1) return;
       fun(n/2);
       fun(n/3);
       fun(n/6);
   }
   ```

   What's the recurrence? Approximate the complexity.
4. Expand (T(n) = T(n-1) + 1) into its explicit sum.

Learning recurrences helps you *see inside* recursion , they turn code into equations.

6. Searching Basics

Before we sort or optimize, we need a way to find things. Searching is one of the most fundamental actions in computing , whether it's looking up a name, finding a key, or checking if something exists.

A search algorithm takes a collection (array, list, tree, etc.) and a target, and returns whether the target is present (and often its position).

Let's begin with two foundational techniques: Linear Search and Binary Search.

1. Linear Search

Linear search is the simplest method:

- Start at the beginning- Check each element in turn- Stop if you find the target It works on any list , sorted or not , but can be slow for large data.

```
int linear_search(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) return i;
    }
    return -1;
}
```

Example: If arr = [2, 4, 6, 8, 10] and key = 6, it finds it at index 2.

Complexity:

- Time: (O(n))- Space: (O(1)) Linear search is simple and guaranteed to find the target if it exists, but slow when lists are large.

2. Binary Search

When the list is sorted, we can do much better. Binary search repeatedly divides the search space in half.

Steps:

1. Check the middle element.
2. If it matches, you're done.
3. If target < mid, search left half.
4. Else search right half.

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

Example: arr = [2, 4, 6, 8, 10], key = 8

- mid = 6 → key > mid → search right half- mid = 8 → found Complexity:
- Time: (O$\log n$)- Space: (O(1)) Binary search is a massive improvement , doubling input only adds one extra step.

3. Recursive Binary Search

Binary search can also be written recursively:

```c
int binary_search_rec(int arr[], int low, int high, int key) {
    if (low > high) return -1;
    int mid = (low + high) / 2;
    if (arr[mid] == key) return mid;
    else if (arr[mid] > key) return binary_search_rec(arr, low, mid - 1, key);
    else return binary_search_rec(arr, mid + 1, high, key);
}
```

Same logic, different structure. Both iterative and recursive forms are equally efficient.

4. Choosing Between Them

Method	Works On	Time	Space	Needs Sorting
Linear Search	Any list	O(n)	O(1)	No
Binary Search	Sorted list	O(log n)	O(1)	Yes

If data is unsorted or very small, linear search is fine. If data is sorted and large, binary search is far superior.

Tiny Code

Compare the steps: For n = 16:

- Linear search → up to 16 comparisons- Binary search → $\log_2 16 = 4$ comparisons That's a huge difference.

Why It Matters

Searching is the core of information retrieval , every database, compiler, and system relies on it.

Understanding simple searches prepares you for:

- Hash tables (constant-time lookups)- Tree searches (ordered structures)- Graph traversals (structured exploration) It's not just about finding values , it's about learning how data structure and algorithm design fit together.

Try It Yourself

1. Write a linear search that returns all indices where a target appears.
2. Modify binary search to return the first occurrence of a target in a sorted array.
3. Compare runtime on arrays of size 10, 100, 1000.
4. What happens if you run binary search on an unsorted list?

Search is the foundation , once you master it, you'll recognize its patterns everywhere.

7. Sorting Basics

Sorting is one of the most studied problems in computer science. Why? Because order matters , it makes searching faster, patterns clearer, and data easier to manage.

A sorting algorithm arranges elements in a specific order (usually ascending or descending). Once sorted, many operations (like binary search, merging, or deduplication) become much simpler.

Let's explore the foundational sorting methods and the principles behind them.

1. What Makes a Sort Algorithm

A sorting algorithm should define:

- Input: A sequence of elements- Output: The same elements, in sorted order- Stability: Keeps equal elements in the same order (important for multi-key sorts)- In-place: Uses only a constant amount of extra space Different algorithms balance speed, memory, and simplicity.

2. Bubble Sort

Idea: Repeatedly "bubble up" the largest element to the end by swapping adjacent pairs.

```
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Each pass moves the largest remaining item to its final position.

- Time: (On^2)- Space: (O(1))- Stable: Yes Simple but inefficient for large data.

3. Selection Sort

Idea: Repeatedly select the smallest element and put it in the correct position.

```
void selection_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) min_idx = j;
        }
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}
```

- Time: (On^2)- Space: (O(1))- Stable: No Fewer swaps, but still quadratic.

4. Insertion Sort

Idea: Build the sorted list one item at a time, inserting each new item in the right place.

```
void insertion_sort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

- Time: (On^2) (best case (O(n)) when nearly sorted)- Space: (O(1))- Stable: Yes Insertion sort is great for small or nearly sorted datasets , used as a base in hybrid sorts like Timsort.

5. Comparing the Basics

Algorithm	Best Case	Average Case	Worst Case	Stable	In-place
Bubble Sort	O(n)	O(n^2)	O(n^2)	Yes	Yes
Selection Sort	O(n^2)	O(n^2)	O(n^2)	No	Yes
Insertion Sort	O(n)	O(n^2)	O(n^2)	Yes	Yes

All three are quadratic in time, but Insertion Sort performs best on small, partially sorted data.

Tiny Code

Quick check with `arr = [5, 3, 4, 1, 2]`:

Insertion Sort (step by step)

- Insert 3 before 5 → [3, 5, 4, 1, 2]- Insert 4 → [3, 4, 5, 1, 2]- Insert 1 → [1, 3, 4, 5, 2]- Insert 2 → [1, 2, 3, 4, 5] Sorted!

Why It Matters

Sorting is a gateway algorithm , it teaches you about iteration, swapping, and optimization.

Efficient sorting is critical for:

- Preprocessing data for binary search- Organizing data for analysis- Building indexes and ranking systems It's the first step toward deeper concepts like divide-and-conquer and hybrid optimization.

Try It Yourself

1. Implement all three: bubble, selection, insertion.
2. Test them on arrays of size 10, 100, 1000 , note timing differences.
3. Try sorting an array that's already sorted. Which one adapts best?
4. Modify insertion sort to sort in descending order.

Sorting may seem simple, but it's a cornerstone , mastering it will shape your intuition for almost every other algorithm.

8. Data Structures Overview

Algorithms and data structures are two sides of the same coin. An algorithm is how you solve a problem. A data structure is where you store and organize data so that your algorithm can work efficiently.

You can think of data structures as containers , each one shaped for specific access patterns, trade-offs, and performance needs. Choosing the right one is often the *key* to designing a fast algorithm.

1. Why Data Structures Matter

Imagine you want to find a book quickly.

- If all books are piled randomly → you must scan every one (O(n)).- If they're sorted on a shelf → you can use binary search (O(log n)).- If you have an index or catalog → you can find it instantly (O(1)). Different structures unlock different efficiencies.

2. The Core Data Structures

Let's walk through the most essential ones:

Type	Description	Key Operations	Typical Use
Array	Fixed-size contiguous memory	Access (O(1)), Insert/Delete (O(n))	Fast index access
Linked List	Sequence of nodes with pointers	Insert/Delete (O(1)), Access (O(n))	Dynamic sequences
Stack	LIFO (last-in, first-out)	push(), pop() in (O(1))	Undo, recursion
Queue	FIFO (first-in, first-out)	enqueue(), dequeue() in (O(1))	Scheduling, buffers

Type	Description	Key Operations	Typical Use
Hash Table	Key-value pairs via hashing	Average (O(1)), Worst (O(n))	Lookup, caching
Heap	Partially ordered tree	Insert ($O\log n$), Extract-Min ($O\log n$)	Priority queues
Tree	Hierarchical structure	Access ($O\log n$) (balanced)	Sorted storage
Graph	Nodes + edges	Traversal (O(V+E))	Networks, paths
Set / Map	Collections of unique keys or key-value pairs	($O\log n$) or (O(1))	Membership tests

Each comes with trade-offs , arrays are fast but rigid, linked lists are flexible but slower to access, and hash tables are lightning-fast but unordered.

3. Abstract Data Types (ADTs)

An ADT defines what operations you can do, not how they're implemented. For example, a Stack ADT promises:

- `push(x)`- `pop()`- `peek()` It can be implemented with arrays or linked lists , the *behavior* stays the same.

Common ADTs:

- Stack- Queue- Deque- Priority Queue- Map / Dictionary This separation of interface and implementation helps design flexible systems.

4. The Right Tool for the Job

Choosing the correct data structure often decides the performance of your algorithm:

Problem	Good Choice	Reason
Undo feature	Stack	LIFO fits history
Scheduling tasks	Queue	FIFO order
Dijkstra's algorithm	Priority Queue	Extract smallest distance
Counting frequencies	Hash Map	Fast key lookup
Dynamic median	Heap + Heap	Balance two halves
Search by prefix	Trie	Fast prefix lookups

Good programmers don't just write code , they pick the right structure.

Tiny Code

Example: comparing array vs linked list

Array:

```c
int arr[5] = {1, 2, 3, 4, 5};
printf("%d", arr[3]); // O(1)
```

Linked List:

```c
struct Node { int val; struct Node* next; };
```

To get the 4th element, you must traverse → (O(n))

Different structures, different access costs.

Why It Matters

Every efficient algorithm depends on the right data structure.

- Searching, sorting, and storing all rely on structure.- Memory layout affects cache performance.- The wrong choice can turn O(1) into O(n^2). Understanding these structures is like knowing the tools in a workshop , once you recognize their shapes, you'll instinctively know which to grab.

Try It Yourself

1. Implement a stack using an array. Then implement it using a linked list.
2. Write a queue using two stacks.
3. Try storing key-value pairs in a hash table (hint: mod by table size).
4. Compare access times for arrays vs linked lists experimentally.

Data structures aren't just storage , they are the *skeletons* your algorithms stand on.

9. Graphs and Trees Overview

Now that you've seen linear structures like arrays and linked lists, it's time to explore non-linear structures , graphs and trees. These are the shapes behind networks, hierarchies, and relationships.

They're everywhere: family trees, file systems, maps, social networks, and knowledge graphs all rely on them.

1. Trees

A tree is a connected structure with no cycles. It's a hierarchy , every node (except the root) has one parent.

- Root: the top node- Child: a node directly connected below- Leaf: a node with no children- Height: the longest path from root to a leaf A binary tree is one where each node has at most two children. A binary search tree (BST) keeps elements ordered:
- Left child < parent < Right child Basic Operations:
- Insert- Search- Delete- Traverse (preorder, inorder, postorder, level-order) Example:

```c
struct Node {
    int val;
    struct Node *left, *right;
};
```

Insert in BST:

```c
struct Node* insert(struct Node* root, int val) {
    if (!root) return newNode(val);
    if (val < root->val) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}
```

Type	Description	Use Case

2. Common Tree Types

Type	Description	Use Case
Binary Tree	Each node has 2 children	General hierarchy
Binary Search Tree (BST)	Left < Root < Right	Ordered data
AVL / Red-Black Tree	Self-balancing BST	Fast search/insert
Heap	Complete binary tree, parent or children	Priority queues
Trie	Tree of characters	Prefix search
Segment Tree	Tree over ranges	Range queries
Fenwick Tree	Tree with prefix sums	Efficient updates

Balanced trees keep height ($O\log n$), guaranteeing fast operations.

3. Graphs

A graph generalizes the idea of trees , now, nodes (vertices) can connect freely. A graph is a set of vertices (V) and edges (E):

$$G = (V, E)$$

Directed vs Undirected

- Directed: edges have direction (A $\to$ B)- Undirected: edges connect both ways (A , B) Weighted vs Unweighted
- Weighted: each edge has a cost- Unweighted: all edges equal Representation:

1. Adjacency Matrix $n \times n$ matrix; entry (i, j) = 1 if edge exists
2. Adjacency List Array of lists: each vertex stores neighbors

Example adjacency list:

```
vector<int> graph[n];
graph[0].push_back(1);
graph[0].push_back(2);
```

4. Common Graph Types

Graph Type	Description	Example
Undirected	Edges without direction	Friendship network
Directed	Arrows indicate direction	Web links
Weighted	Edges have costs	Road network
Cyclic	Contains loops	Task dependencies
Acyclic	No loops	Family tree
DAG (Directed Acyclic Graph)	Directed, no cycles	Scheduling, compilers
Complete	All pairs connected	Dense networks
Sparse	Few edges	Real-world graphs

5. Basic Graph Operations

- Add Vertex / Edge- Traversal: Depth-First Search (DFS), Breadth-First Search (BFS)- Path Finding: Dijkstra, Bellman-Ford- Connectivity: Union-Find, Tarjan (SCC)- Spanning Trees: Kruskal, Prim Each graph problem has its own flavor , from finding shortest paths to detecting cycles.

Tiny Code

Breadth-first search (BFS):

```
void bfs(int start, vector<int> graph[], int n) {
    bool visited[n];
    memset(visited, false, sizeof(visited));
    queue<int> q;
    visited[start] = true;
    q.push(start);
    while (!q.empty()) {
        int node = q.front(); q.pop();
        printf("%d ", node);
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
```

This explores level by level , perfect for shortest paths in unweighted graphs.

Why It Matters

Trees and graphs model relationships and connections, not just sequences. They're essential for:

- Search engines (web graph)- Compilers (syntax trees, dependency DAGs)- AI (state spaces, decision trees)- Databases (indexes, joins, relationships) Understanding them unlocks an entire world of algorithms , from DFS and BFS to Dijkstra, Kruskal, and beyond.

Try It Yourself

1. Build a simple binary search tree and implement inorder traversal.
2. Represent a graph with adjacency lists and print all edges.
3. Write a DFS and BFS for a small graph.
4. Draw a directed graph with a cycle and detect it manually.

Graphs and trees move you beyond linear thinking , they let you explore *connections*, not just collections.

10. Algorithm Design Patterns

By now, you've seen what algorithms *are* and how they're *analyzed*. You've explored searches, sorts, structures, and recursion. The next step is learning patterns , reusable *strategies* that guide how you build new algorithms from scratch.

Just like design patterns in software architecture, algorithmic design patterns give structure to your thinking. Once you recognize them, many problems suddenly feel familiar.

1. Brute Force

Start simple. Try every possibility, pick the best result. Brute force is often your baseline , clear but inefficient.

Example: Find the maximum subarray sum by checking all subarrays.

- Time: (On^2)- Advantage: easy to reason about- Disadvantage: explodes for large input Sometimes, brute force helps you see the structure needed for a better approach.

2. Divide and Conquer

Split the problem into smaller parts, solve each, and combine. Ideal for problems with self-similarity.

Classic examples:

- Merge Sort $\rightarrow$ split and merge- Binary Search $\rightarrow$ halve the search space- Quick Sort $\rightarrow$ partition and sort General form:

$$T(n) = aT(n/b) + f(n)$$

Use recurrence relations and the Master Theorem to analyze them.

3. Greedy

Make the best local decision at each step. Works only when local optimal choices lead to a global optimum.

Examples:

- Activity Selection- Huffman Coding- Dijkstra's (for non-negative weights) Greedy algorithms are simple and fast , when they fit.

4. Dynamic Programming (DP)

When subproblems overlap, store results and reuse them. Think recursion + memory.

Two main styles:

- Top-Down (Memoization): recursive with caching- Bottom-Up (Tabulation): iterative filling table Used in:
- Fibonacci numbers- Knapsack- Longest Increasing Subsequence (LIS)- Matrix Chain Multiplication DP transforms exponential recursion into polynomial time.

5. Backtracking

Explore all possibilities, but prune when constraints fail. It's brute force with early exits.

Perfect for:

- N-Queens- Sudoku- Permutation generation- Subset sums Backtracking builds solutions incrementally, abandoning paths that cannot lead to a valid result.

6. Two Pointers

Move two indices through a sequence to find patterns or meet conditions.

Common use:

- Sorted arrays (sum pairs, partitions)- String problems (palindromes, sliding windows)- Linked lists (slow/fast pointers) Simple, but surprisingly powerful.

7. Sliding Window

Maintain a window over data, expand or shrink it as needed.

Used for:

- Maximum sum subarray (Kadane's algorithm)- Substrings of length k- Longest substring without repeating characters Helps reduce (On^2) to (O(n)) in sequence problems.

8. Binary Search on Answer

Sometimes, the input isn't sorted , but the answer space is. If you can define a function `check(mid)` that's monotonic (true/false changes once), you can apply binary search on possible answers.

Example:

- Minimum capacity to ship packages in D days- Smallest feasible value satisfying a constraint Powerful for optimization under monotonic conditions.

9. Graph-Based

Think in terms of nodes and edges, paths and flows.

Patterns include:

- BFS / DFS (exploration)- Topological Sort (ordering)- Dijkstra / Bellman-Ford (shortest paths)- Union-Find (connectivity)- Kruskal / Prim (spanning trees) Graphs often reveal relationships hidden in data.

10. Meet-in-the-Middle

Split problem into two halves, compute all possibilities for each, and combine efficiently. Used in problems where brute force ($O2^n$) is too large but ($O2^{n/2}$) is manageable.

Example:

- Subset sum (divide into two halves)- Search problems in combinatorics A clever compromise between brute force and efficiency.

Tiny Code

Example: Two Pointers to find pair sum:

```c
int find_pair_sum(int arr[], int n, int target) {
    int i = 0, j = n - 1;
    while (i < j) {
        int sum = arr[i] + arr[j];
        if (sum == target) return 1;
        else if (sum < target) i++;
        else j--;
    }
    return 0;
}
```

Works in (O(n)) for sorted arrays , elegant and fast.

Why It Matters

Patterns are mental shortcuts. They turn “blank page” problems into “I’ve seen this shape before.”

Once you recognize the structure, you can choose a suitable pattern and adapt it. This is how top coders solve complex problems under time pressure , not by memorizing algorithms, but by *seeing patterns.*

Try It Yourself

1. Write a brute-force and a divide-and-conquer solution for maximum subarray sum. Compare speed.
2. Solve the coin change problem using both greedy and DP.
3. Implement N-Queens with backtracking.
4. Use two pointers to find the smallest window with a given sum.
5. Pick a problem you've solved before , can you reframe it using a different design pattern?

The more patterns you practice, the faster you'll map new problems to known strategies , and the more powerful your algorithmic intuition will become.

Chapter 2. Sorting and Searching

11. Elementary Sorting (Bubble, Insertion, Selection)

Before diving into advanced sorts like mergesort or heapsort, it's important to understand the elementary sorting algorithms , the building blocks. They're simple, intuitive, and great for learning how sorting works under the hood.

In this section, we'll cover three classics:

- Bubble Sort - swap adjacent out-of-order pairs- Selection Sort - select the smallest element each time- Insertion Sort - insert elements one by one in order These algorithms share (On^2) time complexity but differ in behavior and stability.

1. Bubble Sort

Idea: Compare adjacent pairs and swap if they're out of order. Repeat until the array is sorted. Each pass "bubbles" the largest element to the end.

Steps:

1. Compare `arr[j]` and `arr[j+1]`
2. Swap if `arr[j] > arr[j+1]`
3. Continue passes until no swaps are needed

Code:

```
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int swapped = 0;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }
        if (!swapped) break;
    }
}
```

Complexity:

- Best: (O(n)) (already sorted)- Worst: (On^2)- Space: (O(1))- Stable: Yes Intuition: Imagine bubbles rising , after each pass, the largest "bubble" settles at the top.

2. Selection Sort

Idea: Find the smallest element and place it at the front.

Steps:

1. For each position `i`, find the smallest element in the remainder of the array
2. Swap it with `arr[i]`

Code:

```
void selection_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}
```

Complexity:

- Best: (On^2)- Worst: (On^2)- Space: (O(1))- Stable: No Intuition: Selection sort "selects" the next correct element and fixes it. It minimizes swaps but still scans all elements.

3. Insertion Sort

Idea: Build a sorted array one element at a time by inserting each new element into its correct position.

Steps:

1. Start from index 1
2. Compare with previous elements
3. Shift elements greater than key to the right
4. Insert key into the correct place

Code:

```
void insertion_sort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Complexity:

- Best: (O(n)) (nearly sorted)- Worst: (On^2)- Space: (O(1))- Stable: Yes Intuition: It's like sorting cards in your hand , take the next card and slide it into the right place.

4. Comparing the Three

Algorithm	Best Case	Average Case	Worst Case	Stable	In-Place	Notes
Bubble Sort	O(n)	O(n^2)	O(n^2)	Yes	Yes	Early exit possible
Selection Sort	O(n^2)	O(n^2)	O(n^2)	No	Yes	Few swaps
Insertion Sort	O(n)	O(n^2)	O(n^2)	Yes	Yes	Great on small or nearly sorted data

Tiny Code

Let's see how insertion sort works on `[5, 3, 4, 1, 2]`:

- Start with 3 → insert before 5 → `[3, 5, 4, 1, 2]`- Insert 4 → `[3, 4, 5, 1, 2]`- Insert 1 → `[1, 3, 4, 5, 2]`- Insert 2 → `[1, 2, 3, 4, 5]` Sorted in five passes.

Why It Matters

Elementary sorts teach you:

- How comparisons and swaps drive order- The trade-off between simplicity and efficiency- How to reason about stability and adaptability While these aren't used for large datasets in practice, they're used *inside* hybrid algorithms like Timsort and IntroSort, which switch to insertion sort for small chunks.

Try It Yourself

1. Implement all three and print the array after each pass.
2. Test on arrays: already sorted, reversed, random, partially sorted.
3. Modify bubble sort to sort descending.
4. Try insertion sort on 10,000 elements and note its behavior.
5. Can you detect when the list is already sorted and stop early?

Start simple. Master these patterns. They'll be your foundation for everything from merge sort to radix sort.

12. Divide-and-Conquer Sorting (Merge, Quick, Heap)

Elementary sorts are great for learning, but their (On^2) runtime quickly becomes a bottleneck. To scale beyond small arrays, we need algorithms that divide problems into smaller parts, sort them independently, and combine the results.

This is the essence of divide and conquer , break it down, solve subproblems, merge solutions. In sorting, this approach yields some of the fastest general-purpose algorithms: Merge Sort, Quick Sort, and Heap Sort.

1. Merge Sort

Idea: Split the array in half, sort each half recursively, then merge the two sorted halves.

Merge sort is stable, works well with linked lists, and guarantees $(On \log n)$ time.

Steps:

1. Divide the array into halves
2. Recursively sort each half
3. Merge two sorted halves into one

Code:

```c
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Complexity:

- Time: $(On\log n)$ (always)- Space: (O(n)) (temporary arrays)- Stable: Yes Merge sort is predictable, making it ideal for external sorting (like sorting data on disk).

2. Quick Sort

Idea: Pick a pivot, partition the array so smaller elements go left and larger go right, then recursively sort both sides.

Quick sort is usually the fastest in practice due to good cache locality and low constant factors.

Steps:

1. Choose a pivot (often middle or random)
2. Partition: move smaller elements to left, larger to right
3. Recursively sort the two partitions

Code:

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            int tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
        }
    }
    int tmp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = tmp;
    return i + 1;
}

void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}
```

Complexity:

- Best / Average: $(On \log n)$- Worst: (On^2) (bad pivot, e.g. sorted input with naive pivot)- Space: $(O\log n)$ (recursion)- Stable: No (unless modified) Quick sort is often used in standard libraries due to its efficiency in real-world workloads.

3. Heap Sort

Idea: Turn the array into a heap, repeatedly extract the largest element, and place it at the end.

A heap is a binary tree where every parent is its children (max-heap).

Steps:

1. Build a max-heap
2. Swap the root (max) with the last element
3. Reduce heap size, re-heapify
4. Repeat until sorted

Code:

```
void heapify(int arr[], int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;
    if (largest != i) {
        int tmp = arr[i]; arr[i] = arr[largest]; arr[largest] = tmp;
        heapify(arr, n, largest);
    }
}

void heap_sort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        int tmp = arr[0]; arr[0] = arr[i]; arr[i] = tmp;
        heapify(arr, i, 0);
    }
}
```

Complexity:

- Time: $(\mathrm{O}n \log n)$- Space: (O(1))- Stable: No Heap sort is reliable and space-efficient but less cache-friendly than quicksort.

4. Comparison

Algorithm	Best Case	Average Case	Worst Case	Space	Stable	Notes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	Predictable, stable
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	No	Fast in practice
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No	In-place, robust

Each one fits a niche:

- Merge Sort → stability and guarantees- Quick Sort → speed and cache performance- Heap Sort → low memory usage and simplicity

Tiny Code

Try sorting `[5, 1, 4, 2, 8]` with merge sort:

1. Split → `[5,1,4]`, `[2,8]`
2. Sort each → `[1,4,5]`, `[2,8]`
3. Merge → `[1,2,4,5,8]`

Each recursive split halves the problem, yielding $(\mathrm{O}\log n)$ depth with (O(n)) work per level.

Why It Matters

Divide-and-conquer sorting is the foundation for efficient order processing. It introduces ideas you'll reuse in:

- Binary search (halving)- Matrix multiplication- Fast Fourier Transform- Dynamic programming These sorts teach how recursion, partitioning, and merging combine into scalable solutions.

Try It Yourself

1. Implement merge sort, quick sort, and heap sort.
2. Test all three on the same random array. Compare runtime.
3. Modify quick sort to use a random pivot.
4. Build a stable version of heap sort.
5. Visualize merge sort's recursion tree and merging process.

Mastering these sorts gives you a template for solving any divide-and-conquer problem efficiently.

13. Counting and Distribution Sorts (Counting, Radix, Bucket)

So far, we've seen comparison-based sorts like merge sort and quicksort. These rely on comparing elements and are bounded by the O(n log n) lower limit for comparisons.

But what if you don't need to compare elements directly , what if they're integers or values from a limited range?

That's where counting and distribution sorts come in. They exploit structure, not just order, to achieve linear-time sorting in the right conditions.

1. Counting Sort

Idea: If your elements are integers in a known range ([0, k)), you can count occurrences of each value, then reconstruct the sorted output.

Counting sort doesn't compare , it counts.

Steps:

1. Find the range of input (max value (k))
2. Count occurrences in a frequency array
3. Convert counts to cumulative counts
4. Place elements into their sorted positions

Code:

```c
void counting_sort(int arr[], int n, int k) {
    int count[k + 1];
    int output[n];
    for (int i = 0; i <= k; i++) count[i] = 0;
    for (int i = 0; i < n; i++) count[arr[i]]++;
    for (int i = 1; i <= k; i++) count[i] += count[i - 1];
    for (int i = n - 1; i >= 0; i--) {
        output[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }
    for (int i = 0; i < n; i++) arr[i] = output[i];
}
```

Example: arr = [4, 2, 2, 8, 3, 3, 1], k = 8 → count = [0,1,2,2,1,0,0,0,1] → cumulative = [0,1,3,5,6,6,6,6,7] → sorted = [1,2,2,3,3,4,8]

Complexity:

- Time: (O(n + k))- Space: (O(k))- Stable: Yes When to use:
- Input is integers- Range (k) not much larger than (n)

2. Radix Sort

Idea: Sort digits one at a time, from least significant (LSD) or most significant (MSD), using a stable sub-sort like counting sort.

Radix sort works best when all elements have fixed-length representations (e.g., integers, strings of equal length).

Steps (LSD method):

1. For each digit position (from rightmost to leftmost)
2. Sort all elements by that digit using a stable sort (like counting sort)

Code:

```
int get_max(int arr[], int n) {
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx) mx = arr[i];
    return mx;
}

void counting_sort_digit(int arr[], int n, int exp) {
    int output[n];
    int count[10] = {0};
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;
    for (int i = 1; i < 10; i++)
```

```
        count[i] += count[i - 1];
    for (int i = n - 1; i >= 0; i--) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

void radix_sort(int arr[], int n) {
    int m = get_max(arr, n);
    for (int exp = 1; m / exp > 0; exp *= 10)
        counting_sort_digit(arr, n, exp);
}
```

Example: arr = [170, 45, 75, 90, 802, 24, 2, 66] → sort by 1s → 10s → 100s → final = [2, 24, 45, 66, 75, 90, 170, 802]

Complexity:

- Time: $(O(d \times (n + b)))$, where
  - (d): number of digits - (b): base (10 for decimal)- Space: $(O(n + b))$- Stable: Yes When to use:
- Fixed-length numbers- Bounded digits (e.g., base 10 or 2)

3. Bucket Sort

Idea: Divide elements into buckets based on value ranges, sort each bucket individually, then concatenate.

Works best when data is uniformly distributed in a known interval.

Steps:

1. Create (k) buckets for value ranges
2. Distribute elements into buckets
3. Sort each bucket (often using insertion sort)
4. Merge buckets

Code:

```
void bucket_sort(float arr[], int n) {
    vector<float> buckets[n];
    for (int i = 0; i < n; i++) {
        int idx = n * arr[i]; // assuming 0 <= arr[i] < 1
        buckets[idx].push_back(arr[i]);
    }
    for (int i = 0; i < n; i++)
        sort(buckets[i].begin(), buckets[i].end());
    int idx = 0;
    for (int i = 0; i < n; i++)
        for (float val : buckets[i])
            arr[idx++] = val;
}
```

Complexity:

- Average: (O(n + k))- Worst: (On^2) (if all fall in one bucket)- Space: (O(n + k))- Stable: Depends on bucket sort method When to use:
- Real numbers uniformly distributed in ([0,1))

4. Comparison

Algorithm	Time	Space	Stable	Type	Best Use
Counting Sort	O(n + k)	O(k)	Yes	Non-comparison	Small integer range
Radix Sort	O(d(n + b))	O(n + b)	Yes	Non-comparison	Fixed-length numbers
Bucket Sort	O(n + k) avg	O(n + k)	Often	Distribution-based	Uniform floats

These algorithms achieve O(n) behavior when assumptions hold , they're specialized but incredibly fast when applicable.

Tiny Code

Let's walk counting sort on `arr = [4, 2, 2, 8, 3, 3, 1]`:

- Count occurrences → [1,2,2,1,0,0,0,1]- Cumulative count → positions- Place elements → [1,2,2,3,3,4,8] Sorted , no comparisons.

Why It Matters

Distribution sorts teach a key insight:

> If you know the structure of your data, you can sort faster than comparison allows.

They show how data properties , range, distribution, digit length , can drive algorithm design.

You'll meet these ideas again in:

- Hashing (bucketing)- Indexing (range partitioning)- Machine learning (binning, histogramming)

Try It Yourself

1. Implement counting sort for integers from 0 to 100.
2. Extend radix sort to sort strings by character.
3. Visualize bucket sort for values between 0 and 1.
4. What happens if you use counting sort on negative numbers? Fix it.
5. Compare counting vs quick sort on small integer arrays.

These are the first glimpses of linear-time sorting , harnessing knowledge about data to break the $(\mathrm{O}n \log n)$ barrier.

14. Hybrid Sorts (IntroSort, Timsort)

In practice, no single sorting algorithm is perfect for all cases. Some are fast on average but fail in worst cases (like Quick Sort). Others are consistent but slow due to overhead (like Merge Sort). Hybrid sorting algorithms combine multiple techniques to get the *best of all worlds* , practical speed, stability, and guaranteed performance.

Two of the most widely used hybrids in modern systems are IntroSort and Timsort , both power the sorting functions in major programming languages.

1. The Idea Behind Hybrid Sorting

Real-world data is messy: sometimes nearly sorted, sometimes random, sometimes pathological. A smart sorting algorithm should adapt to the data.

Hybrids switch between different strategies based on:

- Input size- Recursion depth- Degree of order- Performance thresholds So, the algorithm "introspects" or "adapts" while running.

2. IntroSort

IntroSort (short for *introspective sort*) begins like Quick Sort, but when recursion gets too deep , which means Quick Sort's worst case may be coming , it switches to Heap Sort to guarantee $(On \log n)$ time.

Steps:

1. Use Quick Sort as long as recursion depth $< 2 \log n$
2. If depth exceeds limit $\to$ switch to Heap Sort
3. For very small subarrays $\to$ switch to Insertion Sort

This triple combo ensures:

- Fast average case (Quick Sort)- Guaranteed upper bound (Heap Sort)- Efficiency on small arrays (Insertion Sort) Code Sketch:

```
void intro_sort(int arr[], int n) {
    int depth_limit = 2 * log(n);
    intro_sort_util(arr, 0, n - 1, depth_limit);
}

void intro_sort_util(int arr[], int begin, int end, int depth_limit) {
    int size = end - begin + 1;
    if (size < 16) {
        insertion_sort(arr + begin, size);
        return;
    }
    if (depth_limit == 0) {
        heap_sort_range(arr, begin, end);
        return;
    }
    int pivot = partition(arr, begin, end);
    intro_sort_util(arr, begin, pivot - 1, depth_limit - 1);
    intro_sort_util(arr, pivot + 1, end, depth_limit - 1);
}
```

Complexity:

- Average: $(On \log n)$- Worst: $(On \log n)$- Space: $(O\log n)$- Stable: No (depends on partition scheme) Used in:
- C++ STL's `std::sort`- Many systems where performance guarantees matter

3. Timsort

Timsort is a stable hybrid combining Insertion Sort and Merge Sort. It was designed to handle real-world data, which often has runs (already sorted segments).

Developed by Tim Peters (Python core dev), Timsort is now used in:

- Python's `sorted()` and `.sort()`- Java's `Arrays.sort()` for objects Idea:
- Identify runs , segments already ascending or descending- Reverse descending runs (to make them ascending)- Sort small runs with Insertion Sort- Merge runs with Merge Sort Timsort adapts beautifully to partially ordered data.

Steps:

1. Scan array, detect runs (sequences already sorted)
2. Push runs to a stack
3. Merge runs using a carefully balanced merge strategy

Pseudocode (simplified):

```python
def timsort(arr):
    RUN = 32
    n = len(arr)

    # Step 1: sort small chunks
    for i in range(0, n, RUN):
        insertion_sort(arr, i, min((i + RUN - 1), n - 1))

    # Step 2: merge sorted runs
    size = RUN
    while size < n:
        for start in range(0, n, size * 2):
            mid = start + size - 1
            end = min(start + size * 2 - 1, n - 1)
            merge(arr, start, mid, end)
        size *= 2
```

Complexity:

- Best: (O(n)) (already sorted data)- Average: ($On \log n$)- Worst: ($On \log n$)- Space: (O(n))- Stable: Yes Key Strengths:
- Excellent for real-world, partially sorted data- Stable (keeps equal keys in order)- Optimized merges (adaptive merging)

4. Comparison

Algorithm	Base Methods	Stability	Best	Average	Worst	Real Use
IntroSort	Quick + Heap + Insertion	No	O(n log n)	O(n log n)	O(n log n)	C++ STL
Timsort	Merge + Insertion	Yes	O(n)	O(n log n)	O(n log n)	Python, Java

IntroSort prioritizes performance guarantees. Timsort prioritizes adaptivity and stability.

Both show that "one size fits all" sorting doesn't exist , great systems detect *what's going on* and adapt.

Tiny Code

Suppose we run Timsort on `[1, 2, 3, 7, 6, 5, 8, 9]`:

- Detect runs: `[1,2,3]`, `[7,6,5]`, `[8,9]`- Reverse `[7,6,5]` → `[5,6,7]`- Merge runs → `[1,2,3,5,6,7,8,9]` Efficient because it leverages the existing order.

Why It Matters

Hybrid sorts are the real-world heroes , they combine theory with practice. They teach an important principle:

> When one algorithm's weakness shows up, switch to another's strength.

These are not academic curiosities , they're in your compiler, your browser, your OS, your database. Understanding them means you understand how modern languages optimize fundamental operations.

Try It Yourself

1. Implement IntroSort and test on random, sorted, and reverse-sorted arrays.
2. Simulate Timsort's run detection on nearly sorted input.
3. Compare sorting speed of Insertion Sort vs Timsort for small arrays.
4. Add counters to Quick Sort and see when IntroSort should switch.
5. Explore Python's `sorted()` with different input shapes , guess when it uses merge vs insertion.

Hybrid sorts remind us: good algorithms adapt , they're not rigid, they're smart.

15. Special Sorts (Cycle, Gnome, Comb, Pancake)

Not all sorting algorithms follow the mainstream divide-and-conquer or distribution paradigms. Some were designed to solve niche problems, to illustrate elegant ideas, or simply to experiment with different mechanisms of ordering.

These special sorts, Cycle Sort, Gnome Sort, Comb Sort, and Pancake Sort, are fascinating not because they're the fastest, but because they reveal creative ways to think about permutation, local order, and in-place operations.

1. Cycle Sort

Idea: Minimize the number of writes. Cycle sort rearranges elements into cycles, placing each value directly in its correct position. It performs exactly as many writes as there are misplaced elements, making it ideal for flash memory or systems where writes are expensive.

Steps:

1. For each position `i`, find where `arr[i]` belongs (its rank).
2. If it's not already there, swap it into position.
3. Continue the cycle until the current position is correct.
4. Move to the next index.

Code:

```c
void cycle_sort(int arr[], int n) {
    for (int cycle_start = 0; cycle_start < n - 1; cycle_start++) {
        int item = arr[cycle_start];
        int pos = cycle_start;

        for (int i = cycle_start + 1; i < n; i++)
            if (arr[i] < item) pos++;

        if (pos == cycle_start) continue;

        while (item == arr[pos]) pos++;
        int temp = arr[pos];
        arr[pos] = item;
        item = temp;

        while (pos != cycle_start) {
            pos = cycle_start;
            for (int i = cycle_start + 1; i < n; i++)
```

```
                if (arr[i] < item) pos++;
            while (item == arr[pos]) pos++;
            temp = arr[pos];
            arr[pos] = item;
            item = temp;
        }
    }
}
```

Complexity:

- Time: (On^2)- Writes: minimal (exactly n-c, where c = #cycles)- Stable: No Use Case: When minimizing writes is more important than runtime.

2. Gnome Sort

Idea: A simpler variation of insertion sort. Gnome sort moves back and forth like a "gnome" tidying flower pots: if two adjacent pots are out of order, swap and step back; otherwise, move forward.

Steps:

1. Start at index 1
2. If `arr[i] >= arr[i-1]`, move forward
3. Else, swap and step back (if possible)
4. Repeat until the end

Code:

```
void gnome_sort(int arr[], int n) {
    int i = 1;
    while (i < n) {
        if (i == 0 || arr[i] >= arr[i - 1]) i++;
        else {
            int temp = arr[i]; arr[i] = arr[i - 1]; arr[i - 1] = temp;
            i--;
        }
    }
}
```

Complexity:

- Time: (On^2)- Space: (O(1))- Stable: Yes Use Case: Educational simplicity. It's a readable form of insertion logic without nested loops.

3. Comb Sort

Idea: An improvement over Bubble Sort by introducing a gap between compared elements, shrinking it gradually. By jumping farther apart early, Comb Sort helps eliminate small elements that are “stuck” near the end.

Steps:

1. Start with gap = n
2. On each pass, shrink gap = gap / 1.3
3. Compare and swap items `gap` apart
4. Stop when gap = 1 and no swaps occur

Code:

```
void comb_sort(int arr[], int n) {
    int gap = n;
    int swapped = 1;
    while (gap > 1 || swapped) {
        gap = (gap * 10) / 13;
        if (gap == 9 || gap == 10) gap = 11;
        if (gap < 1) gap = 1;
        swapped = 0;
        for (int i = 0; i + gap < n; i++) {
            if (arr[i] > arr[i + gap]) {
                int temp = arr[i]; arr[i] = arr[i + gap]; arr[i + gap] = temp;
                swapped = 1;
            }
        }
    }
}
```

Complexity:

- Average: $(On \log n)$- Worst: (On^2)- Space: (O(1))- Stable: No Use Case: When a simple, in-place, nearly linear-time alternative to bubble sort is desired.

4. Pancake Sort

Idea: Sort an array using only one operation: flip (reversing a prefix). It’s like sorting pancakes on a plate, flip the stack so the largest pancake goes to the bottom, then repeat for the rest.

Steps:

1. Find the maximum unsorted element
2. Flip it to the front
3. Flip it again to its correct position
4. Reduce the unsorted portion by one

Code:

```
void flip(int arr[], int i) {
    int start = 0;
    while (start < i) {
        int temp = arr[start];
        arr[start] = arr[i];
        arr[i] = temp;
        start++;
        i--;
    }
}

void pancake_sort(int arr[], int n) {
    for (int curr_size = n; curr_size > 1; curr_size--) {
        int mi = 0;
        for (int i = 1; i < curr_size; i++)
            if (arr[i] > arr[mi]) mi = i;
        if (mi != curr_size - 1) {
            flip(arr, mi);
            flip(arr, curr_size - 1);
        }
    }
}
```

Complexity:

- Time: (On^2)- Space: (O(1))- Stable: No Fun Fact: Pancake sort is the only known algorithm whose operations mimic a kitchen utensil, and inspired the Burnt Pancake Problem in combinatorics and genome rearrangement theory.

5. Comparison

Algorithm	Time	Space	Stable	Distinctive Trait
Cycle Sort	O(n^2)	O(1)	No	Minimal writes
Gnome Sort	O(n^2)	O(1)	Yes	Simple insertion-like behavior

Algorithm	Time	Space	Stable	Distinctive Trait
Comb Sort	O(n log n) avg	O(1)	No	Shrinking gap, improved bubble
Pancake Sort	O(n^2)	O(1)	No	Prefix reversals only

Each highlights a different design goal:

- Cycle: minimize writes- Gnome: simplify logic- Comb: optimize comparisons- Pancake: restrict operations

Tiny Code

Example (Pancake Sort on `[3, 6, 1, 9]`):

1. Max = 9 at index 3 → flip(3) → `[9,1,6,3]`
2. flip(3) → `[3,6,1,9]` (9 fixed)
3. Max = 6 → flip(1) → `[6,3,1,9]`
4. flip(2) → `[1,3,6,9]`

Sorted using only flips.

Why It Matters

Special sorts show there's more than one way to think about ordering. They're laboratories for exploring new ideas: minimizing swaps, limiting operations, or optimizing stability. Even if they're not the go-to in production, they deepen your intuition about sorting mechanics.

Try It Yourself

1. Implement each algorithm and visualize their operations step-by-step.
2. Measure how many writes Cycle Sort performs vs. others.
3. Compare Gnome and Insertion sort on nearly sorted arrays.
4. Modify Comb Sort's shrink factor, how does performance change?
5. Write Pancake Sort with printouts of every flip to see the "stack" in motion.

These quirky algorithms prove that sorting isn't just science, it's also art and experimentation.

16. Linear and Binary Search

Searching is the process of finding a target value within a collection of data. Depending on whether the data is sorted or unsorted, you'll use different strategies.

In this section, we revisit two of the most fundamental searching methods , Linear Search and Binary Search , and see how they underpin many higher-level algorithms and data structures.

1. Linear Search

Idea: Check each element one by one until you find the target. This is the simplest possible search and works on unsorted data.

Steps:

1. Start from index 0
2. Compare `arr[i]` with the target
3. If match, return index
4. If end reached, return -1

Code:

```c
int linear_search(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) return i;
    }
    return -1;
}
```

Example: arr = [7, 2, 4, 9, 1], key = 9

- Compare 7, 2, 4, then 9 → found at index 3 Complexity:
- Time: (O(n))- Space: (O(1))- Best case: (O(1)) (first element)- Worst case: (O(n)) Pros:
- Works on any data (sorted or unsorted)- Simple to implement Cons:
- Inefficient on large arrays Use it when data is small or unsorted, or when simplicity matters more than speed.

2. Binary Search

Idea: If the array is sorted, you can repeatedly halve the search space. Compare the middle element to the target , if it's greater, search left; if smaller, search right.

Steps:

1. Find the midpoint
2. If `arr[mid] == key`, done
3. If `arr[mid] > key`, search left
4. If `arr[mid] < key`, search right
5. Repeat until range is empty

Iterative Version:

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

Recursive Version:

```c
int binary_search_rec(int arr[], int low, int high, int key) {
    if (low > high) return -1;
    int mid = (low + high) / 2;
    if (arr[mid] == key) return mid;
    else if (arr[mid] > key)
        return binary_search_rec(arr, low, mid - 1, key);
    else
        return binary_search_rec(arr, mid + 1, high, key);
}
```

Example: arr = [1, 3, 5, 7, 9, 11], key = 7

- mid = 5 $\to$ key > mid $\to$ move right- mid = 7 $\to$ found Complexity:
- Time: ($O \log n$)- Space: (O(1)) (iterative) or ($O \log n$) (recursive)- Best case: (O(1)) (middle element) Requirements:

- Must be sorted- Must have random access (array, not linked list) Pros:
- Very fast for large sorted arrays- Foundation for advanced searches (e.g. interpolation, exponential) Cons:
- Needs sorted data- Doesn't adapt to frequent insertions/deletions

3. Binary Search Variants

Binary search is a *pattern* as much as a single algorithm. You can tweak it to find:

- First occurrence: move left if `arr[mid] == key`- Last occurrence: move right if `arr[mid] == key`- Lower bound: first index key- Upper bound: first index > key Example (Lower Bound):

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

Usage: These variants power functions like `std::lower_bound()` in C++ and binary search trees' lookup logic.

4. Comparison

Algorithm	Works On	Time	Space	Sorted Data Needed	Notes
Linear Search	Any	O(n)	O(1)	No	Best for small/unsorted
Binary Search	Sorted	O(log n)	O(1)	Yes	Fastest on ordered arrays

Binary search trades simplicity for power , once your data is sorted, you unlock sublinear search.

Tiny Code

Compare on array `[2, 4, 6, 8, 10]`, key = 8:

- Linear: 4 steps- Binary: 2 steps This gap grows huge with size , for $n = 10^6$, linear takes up to a million steps, binary about 20.

Why It Matters

These two searches form the foundation of retrieval. Linear search shows brute-force iteration; binary search shows how structure (sorted order) leads to exponential improvement.

From databases to compiler symbol tables to tree lookups, this principle , *divide to search faster* , is everywhere.

Try It Yourself

1. Implement linear and binary search.
2. Count comparisons for (n = 10, 100, 1000).
3. Modify binary search to return the first occurrence of a duplicate.
4. Try binary search on unsorted data , what happens?
5. Combine with sorting: sort array, then search.

Mastering these searches builds intuition for all lookup operations , they are the gateway to efficient data retrieval.

17. Interpolation and Exponential Search

Linear and binary search work well across many scenarios, but they don't take into account how data is distributed. When values are uniformly distributed, we can *estimate* where the target lies, instead of always splitting the range in half. This leads to Interpolation Search, which "jumps" close to where the value should be.

For unbounded or infinite lists, we can't even know the size of the array up front , that's where Exponential Search shines, by quickly expanding its search window before switching to binary search.

Let's dive into both.

1. Interpolation Search

Idea: If data is sorted and uniformly distributed, you can *predict* where a key might be using linear interpolation. Instead of splitting at the middle, estimate the position based on the value's proportion in the range.

Formula:

$$\text{pos} = \text{low} + \frac{(key - arr[low]) \times (high - low)}{arr[high] - arr[low]}$$

This "guesses" where the key lies. If (key = arr[pos]), we're done. Otherwise, adjust `low` or `high` and repeat.

Steps:

1. Compute estimated position `pos`
2. Compare `arr[pos]` with `key`
3. Narrow range accordingly
4. Repeat while `low <= high` and `key` within range

Code:

```c
int interpolation_search(int arr[], int n, int key) {
    int low = 0, high = n - 1;

    while (low <= high && key >= arr[low] && key <= arr[high]) {
        if (low == high) {
            if (arr[low] == key) return low;
            return -1;
        }
        int pos = low + ((double)(key - arr[low]) * (high - low)) / (arr[high] - arr[low]);

        if (arr[pos] == key)
            return pos;
        if (arr[pos] < key)
            low = pos + 1;
        else
            high = pos - 1;
    }
    return -1;
}
```

Example: arr = [10, 20, 30, 40, 50], key = 40 pos = 0 + ((40 - 10) * (4 - 0)) / (50 - 10) = 3 → found at index 3

Complexity:

- Best: (O(1))- Average: ($O\log\log n$) (uniform data)- Worst: (O(n)) (non-uniform or skewed data)- Space: (O(1)) When to Use:
- Data is sorted and nearly uniform- Numeric data where values grow steadily Note: Interpolation search is adaptive , faster when data is predictable, slower when data is irregular.

2. Exponential Search

Idea: When you don't know the array size (e.g., infinite streams, linked data, files), you can't just binary search from 0 to n-1. Exponential search finds a search range dynamically by doubling its step size until it overshoots the target, then does binary search within that range.

Steps:

1. If `arr[0] == key`, return 0
2. Find a range `[bound/2, bound]` such that `arr[bound] >= key`
3. Perform binary search in that range

Code:

```
int exponential_search(int arr[], int n, int key) {
    if (arr[0] == key) return 0;
    int bound = 1;
    while (bound < n && arr[bound] < key)
        bound *= 2;
    int low = bound / 2;
    int high = (bound < n) ? bound : n - 1;
    // Binary search in [low, high]
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key) return mid;
        else if (arr[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

Example: arr = [2, 4, 6, 8, 10, 12, 14, 16], key = 10

- Step: bound = 1 (4), 2 (6), 4 (10 key)- Binary search [2,4] → found Complexity:

- Time: $(O\log i)$, where (i) is index of the target- Space: (O(1))- Best: (O(1)) When to Use:
- Unbounded or streamed data- Unknown array size but sorted order

3. Comparison

Algorithm	Best Case	Average Case	Worst Case	Data Requirement	Notes
Linear Search	O(1)	O(n)	O(n)	Unsorted	Works everywhere
Binary Search	O(1)	O(log n)	O(log n)	Sorted	Predictable halving
Interpolation Search	O(1)	O(log log n)	O(n)	Sorted + Uniform	Adaptive, fast on uniform data
Exponential Search	O(1)	O(log n)	O(log n)	Sorted	Great for unknown size

Interpolation improves on binary *if* data is smooth. Exponential shines when size is unknown.

Tiny Code

Interpolation intuition: If your data is evenly spaced (10, 20, 30, 40, 50), the value 40 should be roughly 75% along. Instead of halving every time, we jump *right near it.* It's data-aware searching.

Exponential intuition: When size is unknown, "expand until you find the wall," then search within.

Why It Matters

These two searches show how context shapes algorithm design:

- *Distribution* (Interpolation Search)- *Boundaries* (Exponential Search) They teach that performance depends not only on structure (sortedness) but also metadata , how much you know about data spacing or limits.

These principles resurface in skip lists, search trees, and probabilistic indexing.

Try It Yourself

1. Test interpolation search on [10, 20, 30, 40, 50] , note how few steps it takes.
2. Try the same on [1, 2, 4, 8, 16, 32, 64] , note slowdown.
3. Implement exponential search and simulate an "infinite" array by stopping at `n`.
4. Compare binary vs interpolation search on random vs uniform data.
5. Extend exponential search to linked lists , how does complexity change?

Understanding these searches helps you tailor lookups to the shape of your data , a key skill in algorithmic thinking.

18. Selection Algorithms (Quickselect, Median of Medians)

Sometimes you don't need to sort an entire array , you just want the k-th smallest (or largest) element. Sorting everything is overkill when you only need one specific rank. Selection algorithms solve this problem efficiently, often in linear time.

They're the backbone of algorithms for median finding, percentiles, and order statistics, and they underpin operations like *pivot selection* in Quick Sort.

1. The Selection Problem

Given an unsorted array of (n) elements and a number (k), find the element that would be at position (k) if the array were sorted.

For example: arr = [7, 2, 9, 4, 6], (k = 3) $\to$ Sorted = [2, 4, 6, 7, 9] $\to$ 3rd smallest = 6

We can solve this without sorting everything.

2. Quickselect

Idea: Quickselect is a selection variant of Quick Sort. It partitions the array around a pivot, but recurses only on the side that contains the k-th element.

It has average-case O(n) time because each partition roughly halves the search space.

Steps:

1. Choose a pivot (random or last element)
2. Partition array into elements < pivot and > pivot
3. Let `pos` be the pivot's index after partition
4. If `pos == k-1` $\to$ done
5. If `pos > k-1` $\to$ recurse left
6. If `pos < k-1` $\to$ recurse right

Code:

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
            i++;
        }
    }
    int temp = arr[i]; arr[i] = arr[high]; arr[high] = temp;
    return i;
}

int quickselect(int arr[], int low, int high, int k) {
    if (low == high) return arr[low];
    int pos = partition(arr, low, high);
    int rank = pos - low + 1;
    if (rank == k) return arr[pos];
    if (rank > k) return quickselect(arr, low, pos - 1, k);
    return quickselect(arr, pos + 1, high, k - rank);
}
```

Example: arr = [7, 2, 9, 4, 6], (k = 3)

- Pivot = 6- Partition → [2, 4, 6, 9, 7], pos = 2- rank = 3 → found (6) Complexity:
- Average: (O(n))- Worst: (On^2) (bad pivots)- Space: (O(1))- In-place When to Use:
- Fast average case- You don't need full sorting Quickselect is used in C++'s `nth_element()` and many median-finding implementations.

3. Median of Medians

Idea: Guarantee worst-case (O(n)) time by choosing a good pivot deterministically.

This method ensures the pivot divides the array into reasonably balanced parts every time.

Steps:

1. Divide array into groups of 5
2. Find the median of each group (using insertion sort)
3. Recursively find the median of these medians → pivot

4. Partition array around this pivot
5. Recurse into the side containing the k-th element

This guarantees at least 30% of elements are eliminated each step → linear time in worst case.

Code Sketch:

```c
int select_pivot(int arr[], int low, int high) {
    int n = high - low + 1;
    if (n <= 5) {
        insertion_sort(arr + low, n);
        return low + n / 2;
    }

    int medians[(n + 4) / 5];
    int i;
    for (i = 0; i < n / 5; i++) {
        insertion_sort(arr + low + i * 5, 5);
        medians[i] = arr[low + i * 5 + 2];
    }
    if (i * 5 < n) {
        insertion_sort(arr + low + i * 5, n % 5);
        medians[i] = arr[low + i * 5 + (n % 5) / 2];
        i++;
    }
    return select_pivot(medians, 0, i - 1);
}
```

You'd then partition around `pivot` and recurse just like Quickselect.

Complexity:

- Worst: (O(n))- Space: (O(1)) (in-place version)- Stable: No (doesn't preserve order) Why It Matters: Median of Medians is slower in practice than Quickselect but provides theoretical guarantees , vital in real-time or critical systems.

4. Special Cases

- Min / Max: trivial , just scan once ((O(n)))- Median: $k = \lceil n/2 \rceil$, can use Quickselect or Median of Medians- Top-k Elements: use partial selection or heaps (k smallest/largest) Example: To get top 5 scores from a million entries, use Quickselect to find 5th largest, then filter threshold.

5. Comparison

Algorithm	Best	Average	Worst	Stable	In-Place	Notes
Quickselect	O(n)	O(n)	O(n^2)	No	Yes	Fast in practice
Median of Medians	O(n)	O(n)	O(n)	No	Yes	Deterministic
Sorting	O(n log n)	O(n log n)	O(n log n)	Depends	Depends	Overkill for single element

Quickselect is fast and simple; Median of Medians is safe and predictable.

Tiny Code

Find 4th smallest in [9, 7, 2, 5, 4, 3]:

- Pivot = 4 → partition [2,3,4,9,7,5]- 4 at position 2 → rank = 3 < 4 → recurse right- New range [9,7,5], (k = 1) → smallest = 5 Result: 5

Why It Matters

Selection algorithms reveal a key insight:

> Sometimes you don't need everything , just what matters.

They form the basis for:

- Median filters in signal processing- Partitioning steps in sorting- k-th order statistics- Robust statistics and quantile computation They embody a "partial work, full answer" philosophy , do exactly enough.

Try It Yourself

1. Implement Quickselect and find k-th smallest for various k.
2. Compare runtime vs full sorting.
3. Modify Quickselect to find k-th largest.
4. Implement Median of Medians pivot selection.
5. Use Quickselect to find median of 1,000 random elements.

Mastering selection algorithms helps you reason about efficiency , you'll learn when to stop sorting and start selecting.

19. Range Searching and Nearest Neighbor

Searching isn't always about finding a single key. Often, you need to find all elements within a given range , or the closest match to a query point.

These problems are central to databases, computational geometry, and machine learning (like k-NN classification). This section introduces algorithms for range queries (e.g. find all values between `L` and `R`) and nearest neighbor searches (e.g. find the point closest to query `q`).

1. Range Searching

Idea: Given a set of data points (1D or multidimensional), quickly report all points within a specified range.

In 1D (simple arrays), range queries can be handled by binary search and prefix sums. In higher dimensions, we need trees designed for efficient spatial querying.

A. 1D Range Query (Sorted Array)

Goal: Find all elements in `[L, R]`.

Steps:

1. Use lower bound to find first element L
2. Use upper bound to find first element > R
3. Output all elements in between

Code (C++-style pseudo):

```
int l = lower_bound(arr, arr + n, L) - arr;
int r = upper_bound(arr, arr + n, R) - arr;
for (int i = l; i < r; i++)
    printf("%d ", arr[i]);
```

Time Complexity:

- Binary search bounds: $(O\log n)$- Reporting results: (O(k)) where (k) = number of elements in range → Total: $(O\log n + k)$

B. Prefix Sum Range Query (For sums)

If you just need the sum (not the actual elements), use prefix sums:

$$\text{prefix}[i] = a_0 + a_1 + \ldots + a_i$$

Then range sum:

$$\text{sum}(L, R) = \text{prefix}[R] - \text{prefix}[L - 1]$$

Code:

```
int prefix[n];
prefix[0] = arr[0];
for (int i = 1; i < n; i++)
    prefix[i] = prefix[i - 1] + arr[i];

int range_sum(int L, int R) {
    return prefix[R] - (L > 0 ? prefix[L - 1] : 0);
}
```

Time: (O(1)) per query after (O(n)) preprocessing.

Used in:

- Databases for fast range aggregation- Fenwick trees, segment trees

C. 2D Range Queries (Rectangular Regions)

For points ((x, y)), queries like:

> "Find all points where $L_x \leq x \leq R_x$ and $L_y \leq y \leq R_y$"

Use specialized structures:

- Range Trees (balanced BSTs per dimension)- Fenwick Trees / Segment Trees (for 2D arrays)- KD-Trees (spatial decomposition) Time: $(O\log^2 n + k)$ typical for 2D Space: $(On \log n)$

2. Nearest Neighbor Search

Idea: Given a set of points, find the one closest to query (q). Distance is often Euclidean, but can be any metric.

Brute Force: Check all points → (O(n)) per query. Too slow for large datasets.

We need structures that let us prune far regions fast.

A. KD-Tree

KD-tree = K-dimensional binary tree. Each level splits points by one coordinate, alternating axes. Used for efficient nearest neighbor search in low dimensions (2D-10D).

Construction:

1. Choose axis = depth % k
2. Sort points by axis
3. Pick median → root
4. Recursively build left and right

Query (Nearest Neighbor):

1. Traverse down tree based on query position
2. Backtrack , check whether hypersphere crosses splitting plane
3. Keep track of best (closest) distance

Complexity:

- Build: ($On \log n$)- Query: ($O\log n$) avg, (O(n)) worst Use Cases:
- Nearest city lookup- Image / feature vector matching- Game AI spatial queries Code Sketch (2D Example):

```
struct Point { double x, y; };

double dist(Point a, Point b) {
    return sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
}
```

(Full KD-tree implementation omitted for brevity , idea is recursive partitioning.)

B. Ball Tree / VP-Tree

For high-dimensional data, KD-trees degrade. Alternatives like Ball Trees (split by hyperspheres) or VP-Trees (Vantage Point Trees) perform better.

They split based on distance metrics, not coordinate axes.

C. Approximate Nearest Neighbor (ANN)

For large-scale, high-dimensional data (e.g. embeddings, vectors):

- Locality Sensitive Hashing (LSH)- HNSW (Hierarchical Navigable Small World Graphs) These trade exactness for speed, common in:
- Vector databases- Recommendation systems- AI model retrieval

3. Summary

Problem	Brute Force	Optimized	Time (Query)	Notes
1D Range Query	Scan O(n)	Binary Search	O(log n + k)	Sorted data
Range Sum	O(n)	Prefix Sum	O(1)	Static data
2D Range Query	O(n)	Range Tree	O($\log^2$ n + k)	Spatial filtering
Nearest Neighbor	O(n)	KD-Tree	O(log n) avg	Exact, low-dim
Nearest Neighbor (high-dim)	O(n)	HNSW / LSH	~O(1)	Approximate

Tiny Code

Simple 1D range query:

```c
int arr[] = {1, 3, 5, 7, 9, 11};
int L = 4, R = 10;
int l = lower_bound(arr, arr + 6, L) - arr;
int r = upper_bound(arr, arr + 6, R) - arr;
for (int i = l; i < r; i++)
    printf("%d ", arr[i]); // 5 7 9
```

Output: `5 7 9`

Why It Matters

Range and nearest-neighbor queries power:

- Databases (SQL range filters, BETWEEN)- Search engines (spatial indexing)- ML (k-NN classifiers, vector similarity)- Graphics / Games (collision detection, spatial queries) These are not just searches , they're geometric lookups, linking algorithms to spatial reasoning.

Try It Yourself

1. Write a function to return all numbers in `[L, R]` using binary search.
2. Build a prefix sum array and answer 5 range-sum queries in O(1).
3. Implement a KD-tree for 2D points and query nearest neighbor.
4. Compare brute-force vs KD-tree search on 1,000 random points.
5. Explore Python's `scipy.spatial.KDTree` or `sklearn.neighbors`.

These algorithms bridge searching with geometry and analytics, forming the backbone of spatial computation.

20. Search Optimizations and Variants

We've explored the main search families , linear, binary, interpolation, exponential , each fitting a different data shape or constraint. Now let's move one step further: optimizing search for performance and adapting it to specialized scenarios.

This section introduces practical variants and enhancements used in real systems, databases, and competitive programming, including jump search, fibonacci search, ternary search, and exponential + binary combinations.

1. Jump Search

Idea: If data is sorted, we can "jump" ahead by fixed steps instead of scanning linearly. It's like hopping through the array in blocks , when you overshoot the target, you step back and linearly search that block.

It strikes a balance between linear and binary search , fewer comparisons without the recursion or halving of binary search.

Steps:

1. Choose jump size = $\sqrt{n}$
2. Jump by blocks until `arr[step] > key`
3. Linear search in previous block

Code:

```
int jump_search(int arr[], int n, int key) {
    int step = sqrt(n);
    int prev = 0;

    while (arr[min(step, n) - 1] < key) {
        prev = step;
        step += sqrt(n);
        if (prev >= n) return -1;
    }

    for (int i = prev; i < min(step, n); i++) {
        if (arr[i] == key) return i;
    }
    return -1;
}
```

Example: arr = [1, 3, 5, 7, 9, 11, 13, 15], key = 11

- step = 2- Jump 5, 7, 9, 11 $\rightarrow$ found Complexity:
- Time: (O$\sqrt{n}$)- Space: (O(1))- Works on sorted data When to Use: For moderately sized sorted lists when you want fewer comparisons but minimal overhead.

2. Fibonacci Search

Idea: Similar to binary search, but it splits the array based on Fibonacci numbers instead of midpoints. This allows using only addition and subtraction (no division), useful on hardware where division is costly.

Also, like binary search, it halves (roughly) the search space each iteration.

Steps:

1. Find the smallest Fibonacci number n
2. Use it to compute probe index
3. Compare and move interval accordingly

Code (Sketch):

```c
int fibonacci_search(int arr[], int n, int key) {
    int fibMMm2 = 0; // (m-2)'th Fibonacci
    int fibMMm1 = 1; // (m-1)'th Fibonacci
    int fibM = fibMMm2 + fibMMm1; // m'th Fibonacci

    while (fibM < n) {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
    }

    int offset = -1;
    while (fibM > 1) {
        int i = min(offset + fibMMm2, n - 1);
        if (arr[i] < key) {
            fibM = fibMMm1;
            fibMMm1 = fibMMm2;
            fibMMm2 = fibM - fibMMm1;
            offset = i;
        } else if (arr[i] > key) {
            fibM = fibMMm2;
            fibMMm1 = fibMMm1 - fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        } else return i;
    }
    if (fibMMm1 && arr[offset + 1] == key)
        return offset + 1;
    return -1;
}
```

Complexity:

- Time: $(O\log n)$- Space: $(O(1))$- Sorted input required Fun Fact: Fibonacci search was originally designed for tape drives , where random access is expensive, and predictable jumps matter.

3. Ternary Search

Idea: When the function or sequence is unimodal (strictly increasing then decreasing), you can locate a maximum or minimum by splitting the range into three parts instead of two.

Used not for discrete lookup but for optimization on sorted functions.

Steps:

1. Divide range into thirds
2. Evaluate at two midpoints `m1`, `m2`
3. Eliminate one-third based on comparison
4. Repeat until range is small

Code:

```
double ternary_search(double low, double high, double (*f)(double)) {
    for (int i = 0; i < 100; i++) {
        double m1 = low + (high - low) / 3;
        double m2 = high - (high - low) / 3;
        if (f(m1) < f(m2))
            low = m1;
        else
            high = m2;
    }
    return (low + high) / 2;
}
```

Example: Find minimum of (f(x) = (x-3)^2) between [0,10]. After iterations, converges to (x 3).

Complexity:

- Time: $O(\log \text{range})$
- Space: $O(1)$
- Works for unimodal functions

Used in:

- Mathematical optimization
- Search-based tuning
- Game AI decision models

4. Binary Search Variants (Review)

Binary search can be tailored to answer richer queries:

- Lower Bound: first index key- Upper Bound: first index > key- Equal Range: range of all equal elements- Rotated Arrays: find element in rotated sorted array- Infinite Arrays: use exponential expansion Rotated Example: arr = [6,7,9,1,3,4], key = 3 → Find pivot, then binary search correct side.

5. Combined Searches

Real systems often chain algorithms:

- Exponential + Binary Search → when bounds unknown- Interpolation + Linear Search → when near target- Jump + Linear Search → hybrid iteration These hybrids use context switching , pick a fast search, then fall back to simple scan in a narrowed window.

6. Summary

Algorithm	Time	Space	Data Requirement	Special Strength
Jump Search	O(√n)	O(1)	Sorted	Fewer comparisons
Fibonacci Search	O(log n)	O(1)	Sorted	Division-free
Ternary Search	O(log range)	O(1)	Unimodal	Optimization
Binary Variants	O(log n)	O(1)	Sorted	Bound finding
Combined Searches	Adaptive	O(1)	Mixed	Practical hybrids

Tiny Code

Jump Search intuition:

```
// Blocks of size sqrt(n)
[1, 3, 5, 7, 9, 11, 13, 15]
Step: 3 → 7 > 6 → search previous block
```

Jumps reduce comparisons dramatically vs linear scan.

Why It Matters

Search optimization is about adapting structure to context. You don't always need a fancy data structure , sometimes a tweak like fixed-step jumping or Fibonacci spacing yields massive gains.

These ideas influence:

- Indexing in databases- Compilers' symbol resolution- Embedded systems with low-level constraints They embody the principle: search smarter, not harder.

Try It Yourself

1. Implement Jump Search and test vs Binary Search on 1M elements.
2. Write a Fibonacci Search , compare steps taken.
3. Use Ternary Search to find min of a convex function.
4. Modify binary search to find element in rotated array.
5. Combine Jump + Linear , how does it behave for small n?

Understanding these variants arms you with flexibility , the heart of algorithmic mastery.

Chapter 3. Data Structures in Actions

21. Arrays, Linked Lists, Stacks, Queues

Every data structure is built on top of a few core foundations , the ones that teach you how data is stored, accessed, and moved. In this section, we'll revisit the essentials: arrays, linked lists, stacks, and queues.

They're simple, but they show you the most important design trade-offs in algorithms:

- Contiguity vs. flexibility- Speed vs. dynamic growth- Last-in-first-out vs. first-in-first-out access

1. Arrays

Idea: A contiguous block of memory storing elements of the same type. Access by index in O(1) time , that's their superpower.

Operations:

- Access `arr[i]`: (O(1))- Update `arr[i]`: (O(1))- Insert at end: (O(1)) (amortized for dynamic arrays)- Insert in middle: (O(n))- Delete: (O(n)) Example:

```c
int arr[5] = {10, 20, 30, 40, 50};
printf("%d", arr[2]); // 30
```

Strengths:

- Fast random access- Cache-friendly (contiguous memory)- Simple, predictable Weaknesses:
- Fixed size (unless using dynamic array)- Costly inserts/deletes Dynamic Arrays: Languages provide resizable arrays (like `vector` in C++ or `ArrayList` in Java) using doubling strategy , when full, allocate new array twice as big and copy. This gives amortized (O(1)) insertion at end.

2. Linked Lists

Idea: A chain of nodes, where each node stores a value and a pointer to the next. No contiguous memory required.

Operations:

- Access: (O(n))- Insert/Delete at head: (O(1))- Search: (O(n)) Example:

```
typedef struct Node {
    int data;
    struct Node* next;
} Node;

Node* head = NULL;
```

Types:

- Singly Linked List: one pointer (next)- Doubly Linked List: two pointers (next, prev)- Circular Linked List: last node points back to first Strengths:
- Dynamic size- Fast insert/delete (no shifting) Weaknesses:
- Slow access- Extra memory for pointers- Poor cache locality Linked lists shine when memory is fragmented or frequent insertions/deletions are needed.

3. Stack

Idea: A Last-In-First-Out (LIFO) structure , the most recently added element is the first to be removed.

Used in:

- Function call stacks- Expression evaluation- Undo operations Operations:
- `push(x)`: add element on top- `pop()`: remove top element- `peek()`: view top element Example (Array-based Stack):

```
#define MAX 100
int stack[MAX], top = -1;

void push(int x) { stack[++top] = x; }
int pop() { return stack[top--]; }
int peek() { return stack[top]; }
```

Complexity: All (O(1)): push, pop, peek

Variants:

- Linked-list-based stack (no fixed size)- Min-stack (tracks minimums) Stacks also appear implicitly , in recursion and backtracking algorithms.

4. Queue

Idea: A First-In-First-Out (FIFO) structure , the first added element leaves first.

Used in:

- Task scheduling- BFS traversal- Producer-consumer pipelines Operations:
- `enqueue(x)`: add to rear- `dequeue()`: remove from front- `front()`: view front Example (Array-based Queue):

```
#define MAX 100
int queue[MAX], front = 0, rear = 0;

void enqueue(int x) { queue[rear++] = x; }
int dequeue() { return queue[front++]; }
```

This simple implementation can waste space. A circular queue fixes that by wrapping indices modulo `MAX`:

```
rear = (rear + 1) % MAX;
```

Complexity: All (O(1)): enqueue, dequeue

Variants:

- Deque (double-ended queue): push/pop from both ends- Priority Queue: dequeue highest priority (not strictly FIFO)

Structure	Access	Insert	Delete	Order	Memory	Notes

5. Comparison

Structure	Access	Insert	Delete	Order	Memory	Notes
Array	O(1)	O(n)	O(n)	Indexed	Contiguous	Fast access
Linked List	O(n)	O(1)*	O(1)*	Sequential	Pointers	Flexible size
Stack	O(1)	O(1)	O(1)	LIFO	Minimal	Call stack, parsing
Queue	O(1)	O(1)	O(1)	FIFO	Minimal	Scheduling, BFS

(* at head or tail with pointer)

Tiny Code

Simple stack example:

```
push(10);
push(20);
printf("%d", pop()); // 20
```

Simple queue example:

```
enqueue(5);
enqueue(8);
printf("%d", dequeue()); // 5
```

These short routines appear in almost every algorithm , from recursion stacks to graph traversals.

Why It Matters

These four structures form the spine of data structures:

- Arrays teach indexing and memory- Linked lists teach pointers and dynamic allocation- Stacks teach recursion and reversal- Queues teach scheduling and order maintenance Every complex structure (trees, heaps, graphs) builds on these.

Master them, and every algorithm will feel more natural.

Try It Yourself

1. Implement a linked list with `insert_front` and `delete_value`.
2. Build a stack and use it to reverse an array.
3. Implement a queue for a round-robin scheduler.
4. Convert infix expression to postfix using a stack.
5. Compare time taken to insert 1000 elements in array vs linked list.

Understanding these foundations gives you the vocabulary of structure , the way algorithms organize their thoughts in memory.

22. Hash Tables and Variants (Cuckoo, Robin Hood, Consistent)

When you need lightning-fast lookups, insertions, and deletions, few data structures match the raw efficiency of a hash table. They're everywhere , from symbol tables and caches to compilers and databases , powering average-case O(1) access.

In this section, we'll unpack how hash tables work, their collision strategies, and explore modern variants like Cuckoo Hashing, Robin Hood Hashing, and Consistent Hashing, each designed to handle different real-world needs.

1. The Core Idea

A hash table maps keys to values using a hash function that transforms the key into an index in an array.

$$\text{index} = h(\text{key}) \bmod \text{table\_size}$$

If no two keys hash to the same index, all operations are (O(1)). But in practice, collisions happen , two keys may map to the same slot , and we must handle them smartly.

2. Collision Resolution Strategies

A. Separate Chaining Each table slot holds a linked list (or dynamic array) of entries with the same hash.

Pros: Simple, handles load factor > 1 Cons: Extra pointers, memory overhead

Code Sketch:

```
typedef struct Node {
    int key, value;
    struct Node* next;
} Node;

Node* table[SIZE];

int hash(int key) { return key % SIZE; }

void insert(int key, int value) {
    int idx = hash(key);
    Node* node = malloc(sizeof(Node));
    node->key = key; node->value = value;
    node->next = table[idx];
    table[idx] = node;
}
```

B. Open Addressing All keys live directly in the table. On collision, find another slot.

Three main strategies:

- Linear probing: try next slot (+1)- Quadratic probing: step size increases quadratically- Double hashing: second hash decides step size Example (Linear Probing):

```
int hash(int key) { return key % SIZE; }
int insert(int key, int value) {
    int idx = hash(key);
    while (table[idx].used)
        idx = (idx + 1) % SIZE;
    table[idx] = (Entry){key, value, 1};
}
```

Load Factor $\alpha = \frac{n}{m}$ affects performance , when too high, rehash to larger size.

3. Modern Variants

Classic hash tables can degrade under heavy collisions. Modern variants reduce probe chains and balance load more evenly.

A. Cuckoo Hashing

Idea: Each key has two possible locations , if both full, evict one ("kick out the cuckoo") and reinsert. Ensures constant lookup , at most two probes.

Steps:

1. Compute two hashes (h_1(key)), (h_2(key))
2. If slot 1 empty → place
3. Else evict occupant, reinsert it using alternate hash
4. Repeat until placed or cycle detected (rehash if needed)

Code Sketch (Conceptual):

```
int h1(int key) { return key % SIZE; }
int h2(int key) { return (key / SIZE) % SIZE; }

void insert(int key) {
    int pos1 = h1(key);
    if (!table1[pos1]) { table1[pos1] = key; return; }
    int displaced = table1[pos1]; table1[pos1] = key;

    int pos2 = h2(displaced);
    if (!table2[pos2]) { table2[pos2] = displaced; return; }
    // continue evicting if needed
}
```

Pros:

- Worst-case O(1) lookup (constant probes)- Predictable latency Cons:
- Rehash needed on insertion failure- More complex logic Used in high-performance caches and real-time systems.

B. Robin Hood Hashing

Idea: Steal slots from richer (closer) keys to ensure fairness. When inserting, if you find someone with smaller probe distance, swap , "steal from the rich, give to the poor."

This balances probe lengths and improves variance and average lookup time.

Key Principle:

$$\text{If new\_probe\_distance} > \text{existing\_probe\_distance} \Rightarrow \text{swap}$$

Code Sketch:

```c
int insert(int key) {
    int idx = hash(key);
    int dist = 0;
    while (table[idx].used) {
        if (table[idx].dist < dist) {
            // swap entries
            Entry tmp = table[idx];
            table[idx] = (Entry){key, dist, 1};
            key = tmp.key;
            dist = tmp.dist;
        }
        idx = (idx + 1) % SIZE;
        dist++;
    }
    table[idx] = (Entry){key, dist, 1};
}
```

Pros:

- Reduced variance- Better performance under high load Cons:
- Slightly slower insertion Used in modern languages like Rust (`hashbrown`) and Swift.

C. Consistent Hashing

Idea: When distributing keys across multiple nodes, you want minimal movement when adding/removing a node. Consistent hashing maps both keys and nodes onto a circular hash ring.

Steps:

1. Hash nodes into a ring
2. Hash keys into same ring
3. Each key belongs to the next node clockwise

When a node is added or removed, only nearby keys move.

Used in:

- Distributed caches (Memcached, DynamoDB)- Load balancing- Sharding in databases Code (Conceptual):

```
Ring: 0 -------------------------------- 2^32
Nodes: N1 at hash("A"), N2 at hash("B")
Key: hash("User42") → assign to next node clockwise
```

Pros:

- Minimal rebalancing- Scalable Cons:
- More complex setup- Requires virtual nodes for even distribution

4. Complexity Overview

Variant	Insert	Search	Delete	Memory	Notes
Chaining	O(1) avg	O(1) avg	O(1) avg	High	Simple, dynamic
Linear Probing	O(1) avg	O(1) avg	O(1) avg	Low	Clustering risk
Cuckoo	O(1)	O(1)	O(1)	Medium	Two tables, predictable
Robin Hood	O(1)	O(1)	O(1)	Low	Balanced probes
Consistent	O(log n)	O(log n)	O(log n)	Depends	Distributed keys

Tiny Code

Simple hash table with linear probing:

```c
#define SIZE 10
int keys[SIZE], values[SIZE], used[SIZE];

int hash(int key) { return key % SIZE; }

void insert(int key, int value) {
    int idx = hash(key);
    while (used[idx]) idx = (idx + 1) % SIZE;
    keys[idx] = key; values[idx] = value; used[idx] = 1;
}
```

Lookup:

```c
int get(int key) {
    int idx = hash(key);
    while (used[idx]) {
        if (keys[idx] == key) return values[idx];
        idx = (idx + 1) % SIZE;
    }
    return -1;
}
```

Why It Matters

Hash tables show how structure and randomness combine for speed. They embody the idea that a good hash function + smart collision handling = near-constant performance.

Variants like Cuckoo and Robin Hood are examples of modern engineering trade-offs , balancing performance, memory, and predictability. Consistent hashing extends these ideas to distributed systems.

Try It Yourself

1. Implement a hash table with chaining and test collision handling.
2. Modify it to use linear probing , measure probe lengths.
3. Simulate Cuckoo hashing with random inserts.
4. Implement Robin Hood swapping logic , observe fairness.
5. Draw a consistent hash ring with 3 nodes and 10 keys , track movement when adding one node.

Once you master these, you'll see hashing everywhere , from dictionaries to distributed databases.

23. Heaps (Binary, Fibonacci, Pairing)

Heaps are priority-driven data structures , they always give you fast access to the most important element, typically the minimum or maximum. They're essential for priority queues, scheduling, graph algorithms (like Dijkstra), and streaming analytics.

In this section, we'll start from the basic binary heap and then explore more advanced ones like Fibonacci and pairing heaps, which trade off simplicity, speed, and amortized guarantees.

1. The Heap Property

A heap is a tree-based structure (often represented as an array) that satisfies:

- Min-Heap: Every node its children- Max-Heap: Every node its children This ensures the root always holds the smallest (or largest) element.

Complete Binary Tree: All levels filled except possibly the last, which is filled left to right.

Example (Min-Heap):

```
       2
     /   \
    4     5
   / \   /
  9  10 15
```

Here, the smallest element (2) is at the root.

2. Binary Heap

Storage: Stored compactly in an array. For index `i` (0-based):

- Parent = `(i - 1) / 2`- Left child = `2i + 1`- Right child = `2i + 2` Operations:

Operation	Description	Time
`push(x)`	Insert element	$(O\log n)$
`pop()`	Remove root	$(O\log n)$
`peek()`	Get root	(O(1))
`heapify()`	Build heap	(O(n))

A. Insertion (Push)

Insert at the end, then bubble up until heap property is restored.

Code:

```
void push(int heap[], int *n, int x) {
    int i = (*n)++;
    heap[i] = x;
    while (i > 0 && heap[(i - 1)/2] > heap[i]) {
        int tmp = heap[i];
        heap[i] = heap[(i - 1)/2];
        heap[(i - 1)/2] = tmp;
        i = (i - 1) / 2;
    }
}
```

B. Removal (Pop)

Replace root with last element, then bubble down (heapify).

Code:

```
void heapify(int heap[], int n, int i) {
    int smallest = i, l = 2*i + 1, r = 2*i + 2;
    if (l < n && heap[l] < heap[smallest]) smallest = l;
    if (r < n && heap[r] < heap[smallest]) smallest = r;
    if (smallest != i) {
        int tmp = heap[i]; heap[i] = heap[smallest]; heap[smallest] = tmp;
        heapify(heap, n, smallest);
    }
}
```

Pop:

```
int pop(int heap[], int *n) {
    int root = heap[0];
    heap[0] = heap[--(*n)];
    heapify(heap, *n, 0);
    return root;
}
```

C. Building a Heap

Heapify bottom-up from last non-leaf: (O(n))

```
for (int i = n/2 - 1; i >= 0; i--)
    heapify(heap, n, i);
```

D. Applications

- Heapsort: Repeatedly pop min (O(n log n))- Priority Queue: Fast access to smallest/largest- Graph Algorithms: Dijkstra, Prim- Streaming: Median finding using two heaps

3. Fibonacci Heap

Idea: A heap optimized for algorithms that do many decrease-key operations (like Dijkstra's). It stores a collection of trees with lazy merging, giving amortized bounds:

Operation	Amortized Time
Insert	(O(1))
Find-Min	(O(1))
Extract-Min	$(\mathrm{O}\log n)$
Decrease-Key	(O(1))
Merge	(O(1))

It achieves this by delaying structural fixes until absolutely necessary (using potential method in amortized analysis).

Structure:

- A circular linked list of roots- Each node can have multiple children- Consolidation on extract-min ensures minimal degree duplication Used in theoretical optimizations where asymptotic complexity matters (e.g. Dijkstra in $(\mathrm{O}E + V\log V)$ vs $(\mathrm{O}E\log V)$).

4. Pairing Heap

Idea: A simpler, practical alternative to Fibonacci heaps. Self-adjusting structure using a tree with multiple children.

Operations:

- Insert: (O(1))- Extract-Min: $(\mathrm{O}\log n)$ amortized- Decrease-Key: $(\mathrm{O}\log n)$ amortized Steps:

- `merge` two heaps: attach one as child of the other- `extract-min`: remove root, merge children in pairs, then merge all results Why It's Popular:
- Easier to implement- Great real-world performance- Used in functional programming and priority schedulers

5. Comparison

Heap Type	Insert	Extract-Min	Decrease-Key	Merge	Simplicity	Use Case
Binary Heap	O(log n)	O(log n)	O(log n)	O(n)	Easy	General-purpose
Fibonacci Heap	O(1)	O(log n)	O(1)	O(1)	Complex	Theoretical optimality
Pairing Heap	O(1)	O(log n)	O(log n)	O(1)	Moderate	Practical alternative

Tiny Code

Binary Heap Demo:

```
int heap[100], n = 0;
push(heap, &n, 10);
push(heap, &n, 4);
push(heap, &n, 7);
printf("%d ", pop(heap, &n)); // 4
```

Output: `4`

Why It Matters

Heaps show how to prioritize elements dynamically. From sorting to scheduling, they're the backbone of many "choose the best next" algorithms. Variants like Fibonacci and Pairing Heaps demonstrate how amortized analysis can unlock deeper efficiency , crucial in graph theory and large-scale optimization.

Try It Yourself

1. Implement a binary min-heap with `push`, `pop`, and `peek`.
2. Use a heap to sort a list (Heapsort).
3. Build a priority queue for task scheduling.
4. Study how Dijkstra changes when replacing arrays with heaps.
5. Explore Fibonacci heap pseudo-code , trace `decrease-key`.

Mastering heaps gives you a deep sense of priority-driven design , how to keep "the best" element always within reach.

24. Balanced Trees (AVL, Red-Black, Splay, Treap)

Unbalanced trees can degrade into linear lists, turning your beautiful (O$\log n$) search into a sad (O(n)) crawl. Balanced trees solve this , they keep the height logarithmic, guaranteeing fast lookups, insertions, and deletions.

In this section, you'll learn how different balancing philosophies work , AVL (strict balance), Red-Black (relaxed balance), Splay (self-adjusting), and Treap (randomized balance).

1. The Idea of Balance

For a binary search tree (BST):

$$\text{height} = O(\log n)$$

only if it's balanced , meaning the number of nodes in left and right subtrees differ by a small factor.

Unbalanced BST (bad):

```
1
 \
  2
   \
    3
```

Balanced BST (good):

```
  2
 / \
1   3
```

Balance ensures efficient:

- `search(x)` $\to (O\log n)$- `insert(x)` $\to (O\log n)$- `delete(x)` $\to (O\log n)$

2. AVL Tree (Adelson-Velsky & Landis)

Invented in 1962, AVL is the first self-balancing BST. It enforces strict balance:

$$|\text{height}(\text{left}) - \text{height}(\text{right})| \leq 1$$

Whenever this condition breaks, rotations fix it.

Rotations:

- LL (Right Rotation): imbalance on left-left- RR (Left Rotation): imbalance on right-right- LR / RL: double rotation cases Code (Rotation Example):

```
Node* rotateRight(Node* y) {
    Node* x = y->left;
    Node* T = x->right;
    x->right = y;
    y->left = T;
    return x;
}
```

Height & Balance Factor:

```
int height(Node* n) { return n ? n->h : 0; }
int balance(Node* n) { return height(n->left) - height(n->right); }
```

Properties:

- Strict height bound: $(O\log n)$- More rotations (slower insertions)- Excellent lookup speed Used when lookups > updates (databases, indexing).

3. Red-Black Tree

Idea: A slightly looser balance for faster insertions. Each node has a color (Red/Black) with these rules:

1. Root is black
2. Red node's children are black
3. Every path has same number of black nodes
4. Null nodes are black

Balance through color flips + rotations

Compared to AVL:

- Fewer rotations (faster insert/delete)- Slightly taller (slower lookup)- Simpler amortized balance Used in:

- C++ `std::map`, `std::set`- Java `TreeMap`, Linux scheduler Complexity: All major operations $(O\log n)$

4. Splay Tree

Idea: Bring recently accessed node to root via splaying (rotations). It adapts to access patterns , the more you access a key, the faster it becomes.

Splaying Steps:

- Zig: one rotation (root child)- Zig-Zig: two rotations (same side)- Zig-Zag: two rotations (different sides) Code (Conceptual):

```
Node* splay(Node* root, int key) {
    if (!root || root->key == key) return root;
    if (key < root->key) {
        if (!root->left) return root;
        // splay in left subtree
        if (key < root->left->key)
            root->left->left = splay(root->left->left, key),
            root = rotateRight(root);
        else if (key > root->left->key)
            root->left->right = splay(root->left->right, key),
            root->left = rotateLeft(root->left);
        return rotateRight(root);
    } else {
        if (!root->right) return root;
        // symmetric
    }
}
```

Why It's Cool: No strict balance, but amortized ($O\log n$). Frequently accessed elements stay near top.

Used in self-adjusting caches, rope data structures, memory allocators.

5. Treap (Tree + Heap)

Idea: Each node has two keys:

- BST key → order property- Priority → heap property Insertion = normal BST insert + heap fix via rotation.

Balance comes from randomization , random priorities ensure expected ($O\log n$) height.

Code Sketch:

```c
typedef struct Node {
    int key, priority;
    struct Node *left, *right;
} Node;

Node* insert(Node* root, int key) {
    if (!root) return newNode(key, rand());
    if (key < root->key) root->left = insert(root->left, key);
    else root->right = insert(root->right, key);

    if (root->left && root->left->priority > root->priority)
        root = rotateRight(root);
    if (root->right && root->right->priority > root->priority)
        root = rotateLeft(root);
    return root;
}
```

Advantages:

- Simple logic- Random balancing- Expected ($O\log n$) Used in randomized algorithms and functional programming.

6. Comparison

Tree	Balance Type	Rotations	Height	Insert/Delete	Lookup	Notes
AVL	Strict	More	$(O\log n)$	Medium	Fast	Lookup-heavy
Red-Black	Relaxed	Fewer	$(O\log n)$	Fast	Medium	Library std
Splay	Adaptive	Variable	Amortized $(O\log n)$	Fast	Fast (amortized)	Access patterns
Treap	Randomized	Avg few	$(O\log n)$ expected	Simple	Simple	Probabilistic

Tiny Code

AVL Insert (Skeleton):

```c
Node* insert(Node* root, int key) {
    if (!root) return newNode(key);
    if (key < root->key) root->left = insert(root->left, key);
    else root->right = insert(root->right, key);
    root->h = 1 + max(height(root->left), height(root->right));
    int b = balance(root);
    if (b > 1 && key < root->left->key) return rotateRight(root);
    if (b < -1 && key > root->right->key) return rotateLeft(root);
    // other cases...
    return root;
}
```

Why It Matters

Balanced trees guarantee predictable performance under dynamic updates. Each variant represents a philosophy:

- AVL: precision- Red-Black: practicality- Splay: adaptability- Treap: randomness Together, they teach one core idea , keep height in check, no matter the operations.

Try It Yourself

1. Implement an AVL tree and visualize rotations.
2. Insert keys [10, 20, 30, 40, 50] and trace Red-Black color changes.
3. Splay after each access , see which keys stay near top.
4. Build a Treap with random priorities , measure average height.
5. Compare performance of BST vs AVL on sorted input.

Balanced trees are the architects of order , always keeping chaos one rotation away.

25. Segment Trees and Fenwick Trees

When you need to answer range queries quickly (like sum, min, max) and support updates to individual elements, simple prefix sums won't cut it anymore.

You need something smarter , data structures that can divide and conquer over ranges, updating and combining results efficiently.

That's exactly what Segment Trees and Fenwick Trees (Binary Indexed Trees) do:

- Query over a range in $(O \log n)$- Update elements in $(O \log n)$ They're the backbone of competitive programming, signal processing, and database analytics.

1. The Problem

Given an array `A[0..n-1]`, support:

1. `update(i, x)` $\rightarrow$ change `A[i]` to `x`
2. `query(L, R)` $\rightarrow$ compute sum (or min, max) of `A[L..R]`

Naive approach:

- Update: (O(1))- Query: (O(n)) Prefix sums fix one but not both. Segment and Fenwick trees fix both.

2. Segment Tree

Idea: Divide the array into segments (intervals) recursively. Each node stores an aggregate (sum, min, max) of its range. You can combine child nodes to get any range result.

Structure (Sum Example):

```
          [0,7] sum=36
         /            \
   [0,3]=10         [4,7]=26
   /     \            /      \
[0,1]=3 [2,3]=7  [4,5]=11  [6,7]=15
```

Each node represents a range [L,R]. Leaf nodes = single elements.

A. Build

Recursive Construction: Time: (O(n))

```
void build(int node, int L, int R) {
    if (L == R) tree[node] = arr[L];
    else {
        int mid = (L + R) / 2;
        build(2*node, L, mid);
        build(2*node+1, mid+1, R);
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}
```

B. Query (Range Sum)

Query [l, r] recursively:

- If current range [L, R] fully inside [l, r], return node value- If disjoint, return 0- Else combine children

```
int query(int node, int L, int R, int l, int r) {
    if (r < L || R < l) return 0;
    if (l <= L && R <= r) return tree[node];
    int mid = (L + R) / 2;
    return query(2*node, L, mid, l, r)
         + query(2*node+1, mid+1, R, l, r);
}
```

C. Update

Change `arr[i] = x` and update tree nodes covering `i`.

```
void update(int node, int L, int R, int i, int x) {
    if (L == R) tree[node] = x;
    else {
        int mid = (L + R)/2;
        if (i <= mid) update(2*node, L, mid, i, x);
        else update(2*node+1, mid+1, R, i, x);
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}
```

Complexities:

- Build: (O(n))- Query: ($O\log n$)- Update: ($O\log n$)- Space: (O(4n))

D. Variants

Segment trees are flexible:

- Range minimum/maximum- Range GCD- Lazy propagation → range updates- 2D segment tree for grids

3. Fenwick Tree (Binary Indexed Tree)

Idea: Stores cumulative frequencies using bit manipulation. Each node covers a range size = LSB(index).

Simpler, smaller, but supports only associative ops (sum, xor, etc.)

Indexing:

- Parent: `i + (i & -i)`- Child: `i - (i & -i)` Build: Initialize with zero, then add elements one by one.

Add / Update:

```
void add(int i, int x) {
    for (; i <= n; i += i & -i)
        bit[i] += x;
}
```

Prefix Sum:

```
int sum(int i) {
    int res = 0;
    for (; i > 0; i -= i & -i)
        res += bit[i];
    return res;
}
```

Range Sum [L, R]:

$$\operatorname{sum}(R) - \operatorname{sum}(L-1)$$

Complexities:

- Build: $(On \log n)$- Query: $(O \log n)$- Update: $(O \log n)$- Space: (O(n))

4. Comparison

Feature	Segment Tree	Fenwick Tree
Space	O(4n)	O(n)
Build	O(n)	O(n log n)
Query	O(log n)	O(log n)
Update	O(log n)	O(log n)
Range Update	With Lazy	Tricky
Range Query	Flexible	Sum/XOR only
Implementation	Moderate	Simple

5. Applications

- Sum / Min / Max / XOR queries- Frequency counts- Inversions counting- Order statistics- Online problems where array updates over time Used in:
- Competitive programming- Databases (analytics on changing data)- Time series queries- Games (damage/range updates)

Tiny Code

Fenwick Tree Example:

```
int bit[1001], n;

void update(int i, int val) {
    for (; i <= n; i += i & -i)
        bit[i] += val;
}

int query(int i) {
    int res = 0;
    for (; i > 0; i -= i & -i)
        res += bit[i];
    return res;
}

// range sum
int range_sum(int L, int R) { return query(R) - query(L - 1); }
```

Why It Matters

Segment and Fenwick trees embody divide-and-conquer over data , balancing dynamic updates with range queries. They're how modern systems aggregate live data efficiently.

They teach a powerful mindset:

> "If you can split a problem, you can solve it fast."

Try It Yourself

1. Build a segment tree for sum queries.
2. Add range minimum queries (RMQ).
3. Implement a Fenwick tree , test with prefix sums.
4. Solve: number of inversions in array using Fenwick tree.
5. Add lazy propagation to segment tree for range updates.

Once you master these, range queries will never scare you again , you'll slice through them in logarithmic time.

26. Disjoint Set Union (Union-Find)

Many problems involve grouping elements into sets and efficiently checking whether two elements belong to the same group , like connected components in a graph, network connectivity, Kruskal's MST, or even social network clustering.

For these, the go-to structure is the Disjoint Set Union (DSU), also called Union-Find. It efficiently supports two operations:

1. `find(x)` → which set does `x` belong to?
2. `union(x, y)` → merge the sets containing `x` and `y`.

With path compression and union by rank, both operations run in near-constant time, specifically (O((n))), where α is the inverse Ackermann function (practically 4).

1. The Problem

Suppose you have (n) elements initially in separate sets. Over time, you want to:

- Merge two sets- Check if two elements share the same set Example:

```
Sets: {1}, {2}, {3}, {4}, {5}
Union(1,2) → {1,2}, {3}, {4}, {5}
Union(3,4) → {1,2}, {3,4}, {5}
Find(2) == Find(1)? Yes
Find(5) == Find(3)? No
```

2. Basic Implementation

Each element has a parent pointer. Initially, every node is its own parent.

Parent array representation:

```
int parent[N];

void make_set(int v) {
    parent[v] = v;
}

int find(int v) {
    if (v == parent[v]) return v;
    return find(parent[v]);
}
```

```
void union_sets(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b)
        parent[b] = a;
}
```

This works, but deep trees can form , making `find` slow. We fix that with path compression.

3. Path Compression

Every time we call `find(v)`, we make all nodes along the path point directly to the root. This flattens the tree dramatically.

Optimized Find:

```
int find(int v) {
    if (v == parent[v]) return v;
    return parent[v] = find(parent[v]);
}
```

So next time, lookups will be (O(1)) for those nodes.

4. Union by Rank / Size

When merging, always attach the smaller tree to the larger to keep depth small.

Union by Rank:

```
int parent[N], rank[N];

void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

void union_sets(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        if (rank[a] < rank[b])
```

```
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
```

Union by Size (Alternative): Track size of each set and attach smaller to larger.

```
int size[N];
void union_sets(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        if (size[a] < size[b]) swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}
```

5. Complexity

With both path compression and union by rank, all operations are effectively constant time:

$$O(\alpha(n)) \approx O(1)$$

For all practical (n), ((n) 4).

Operation	Time
Make set	O(1)
Find	O((n))
Union	O((n))

6. Applications

- Graph Connectivity: determine connected components- Kruskal's MST: add edges, avoid cycles- Dynamic connectivity- Image segmentation- Network clustering- Cycle detection in undirected graphs Example: Kruskal's Algorithm

```
sort(edges.begin(), edges.end());
for (edge e : edges)
    if (find(e.u) != find(e.v)) {
        union_sets(e.u, e.v);
        mst_weight += e.w;
    }
```

7. Example

```
int parent[6], rank[6];

void init() {
    for (int i = 1; i <= 5; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}

int main() {
    init();
    union_sets(1, 2);
    union_sets(3, 4);
    union_sets(2, 3);
    printf("%d\n", find(4)); // prints representative of {1,2,3,4}
}
```

Result: `{1,2,3,4}`, `{5}`

8. Visualization

```
Before compression:
1
 \
  2
   \
    3

After compression:
1
```

```
2
3
```

Every `find` call makes future queries faster.

9. Comparison

Variant	Find	Union	Notes
Basic	O(n)	O(n)	Deep trees
Path Compression	O((n))	O((n))	Very fast
+ Rank / Size	O((n))	O((n))	Balanced
Persistent DSU	O(log n)	O(log n)	Undo/rollback support

Tiny Code

Full DSU with path compression + rank:

```
int parent[1000], rank[1000];

void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find(int v) {
    if (v != parent[v])
        parent[v] = find(parent[v]);
    return parent[v];
}

void union_sets(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        if (rank[a] < rank[b]) swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
```

Why It Matters

Union-Find embodies structural sharing and lazy optimization , you don't balance eagerly, but just enough. It's one of the most elegant demonstrations of how constant-time algorithms are possible through clever organization.

It teaches a key algorithmic lesson:

> "Work only when necessary, and fix structure as you go."

Try It Yourself

1. Implement DSU and test `find`/`union`.
2. Build a program that counts connected components.
3. Solve Kruskal's MST using DSU.
4. Add `get_size(v)` to return component size.
5. Try rollback DSU (keep stack of changes).

Union-Find is the quiet powerhouse behind many graph and connectivity algorithms , simple, fast, and deeply elegant.

27. Probabilistic Data Structures (Bloom, Count-Min, HyperLogLog)

When you work with massive data streams , billions of elements, too big for memory , you can't store everything. But what if you don't need *perfect* answers, just *fast and tiny approximate ones*?

That's where probabilistic data structures shine. They trade a bit of accuracy for huge space savings and constant-time operations.

In this section, we'll explore three of the most famous:

- Bloom Filters → membership queries- Count-Min Sketch → frequency estimation- HyperLogLog → cardinality estimation Each of them answers "How likely is X?" or "How many?" efficiently , perfect for modern analytics, caching, and streaming systems.

1. Bloom Filter , "Is this element probably in the set?"

A Bloom filter answers:

> "Is `x` in the set?" with either maybe yes or definitely no.

No false negatives, but *some* false positives.

A. Idea

Use an array of bits (size `m`), all initialized to 0. Use k different hash functions.

To insert an element:

1. Compute k hashes: (h_1(x), h_2(x), …, h_k(x))
2. Set each bit position $b_i = 1$

To query an element:

1. Compute same k hashes
2. If all bits are 1 $\to$ maybe yes
3. If any bit is 0 $\to$ definitely no

B. Example

Insert `dog`:

- (h_1(dog)=2, h_2(dog)=5, h_3(dog)=9) Set bits 2, 5, 9 $\to$ 1

Check `cat`:

- If any hash bit $= 0 \to$ not present

C. Complexity

Operation	Time	Space	Accuracy
Insert	O(k)	O(m)	Tunable
Query	O(k)	O(m)	False positives

False positive rate   ($1 - e^{-kn/m}$^k)

Choose `m` and `k` based on expected `n` and acceptable error.

D. Code

```
#define M 1000
int bitset[M];

int hash1(int x) { return (x * 17) % M; }
int hash2(int x) { return (x * 31 + 7) % M; }

void add(int x) {
    bitset[hash1(x)] = 1;
    bitset[hash2(x)] = 1;
}

bool contains(int x) {
    return bitset[hash1(x)] && bitset[hash2(x)];
}
```

Used in:

- Caches (check before disk lookup)- Spam filters- Databases (join filtering)- Blockchain and peer-to-peer networks

2. Count-Min Sketch , "How often has this appeared?"

Tracks frequency counts in a stream, using sub-linear memory.

Instead of a full table, it uses a 2D array of counters, each row hashed with a different hash function.

A. Insert

For each row `i`:

- Compute hash (h_i(x))- Increment `count[i][h_i(x)]++` #### B. Query

For element `x`:

- Compute all (h_i(x))- Take `min(count[i][h_i(x)])` across rows → gives an upper-bounded estimate of true frequency

C. Code

```
#define W 1000
#define D 5
int count[D][W];

int hash(int i, int x) {
    return (x * (17*i + 3)) % W;
}

void add(int x) {
    for (int i = 0; i < D; i++)
        count[i][hash(i, x)]++;
}

int query(int x) {
    int res = INT_MAX;
    for (int i = 0; i < D; i++)
        res = min(res, count[i][hash(i, x)]);
    return res;
}
```

D. Complexity

Operation	Time	Space
Insert	O(D)	O(W×D)
Query	O(D)	O(W×D)

Error controlled by:

$$\varepsilon = \frac{1}{W}, \quad \delta = 1 - e^{-D}$$

Used in:

- Frequency counting in streams- Hot-key detection- Network flow analysis- Trending topics

3. HyperLogLog , "How many unique items?"

Estimates cardinality (number of distinct elements) with very small memory (~1.5 KB for millions).

A. Idea

Hash each element uniformly → 32-bit value. Split hash into:

- Prefix bits → bucket index- Suffix bits → count leading zeros Each bucket stores the max leading zero count seen. At the end, use harmonic mean of counts to estimate distinct values.

B. Formula

$$E = \alpha_m \cdot m^2 \cdot \left(\sum_{i=1}^{m} 2^{-M[i]} \right)^{-1}$$

where `M[i]` is the zero count in bucket `i`, and α_m is a correction constant.

Accuracy: ~1.04 / √m

C. Complexity

Operation	Time	Space	Error
Add	O(1)	O(m)	~1.04/√m
Merge	O(m)	O(m)	,

Used in:

- Web analytics (unique visitors)- Databases (`COUNT DISTINCT`)- Distributed systems (mergeable estimates)

4. Comparison

Structure	Purpose	Query	Memory	Error	Notes
Bloom	Membership	O(k)	Tiny	False positives	No deletions
Count-Min	Frequency	O(D)	Small	Overestimate	Streaming counts
Hyper-LogLog	Cardinality	O(1)	Very small	~1%	Mergeable

Tiny Code

Bloom Filter Demo:

```
add(42);
add(17);
printf("%d\n", contains(42)); // 1 (maybe yes)
printf("%d\n", contains(99)); // 0 (definitely no)
```

Why It Matters

Probabilistic data structures show how approximation beats impossibility when resources are tight. They make it feasible to process massive streams in real time, when storing everything is impossible.

They teach a deeper algorithmic truth:

> "A bit of uncertainty can buy you a world of scalability."

Try It Yourself

1. Implement a Bloom filter with 3 hash functions.
2. Measure false positive rate for 10K elements.
3. Build a Count-Min Sketch and test frequency estimation.
4. Approximate unique elements using HyperLogLog logic.
5. Explore real-world systems: Redis (Bloom/CM Sketch), PostgreSQL (HyperLogLog).

These tiny probabilistic tools are how big data becomes tractable.

28. Skip Lists and B-Trees

When you want fast search, insert, and delete but need a structure that's easier to code than trees or optimized for disk and memory blocks, two clever ideas step in:

- Skip Lists $\rightarrow$ randomized, layered linked lists that behave like balanced BSTs- B-Trees $\rightarrow$ multi-way trees that minimize disk I/O and organize large data blocks Both guarantee (O$\log n$) operations, but they shine in very different environments , Skip Lists in-memory, B-Trees on disk.

1. Skip Lists

Invented by: William Pugh (1990) Goal: Simulate binary search using linked lists with probabilistic shortcuts.

A. Idea

A skip list is a stack of linked lists, each level skipping over more elements.

Example:

```
Level 3:            > 50
Level 2:     > 10    > 30    > 50
Level 1:  5   > 10   > 20   > 30   > 40   > 50
```

Higher levels are sparser and let you "skip" large chunks of the list.

You search top-down:

- Move right while next target- Drop down when you can't go further This mimics binary search , logarithmic layers, logarithmic hops.

B. Construction

Each inserted element is given a random height, with geometric distribution:

- Level 1 (base) always exists- Level 2 with probability ½- Level 3 with ¼, etc. Expected total nodes = 2n, Expected height = (O$\log n$)

C. Operations

Operation	Time	Space	Notes
Search	(O$\log n$)	O(n)	Randomized balance
Insert	(O$\log n$)	O(n)	Rebuild towers
Delete	(O$\log n$)	O(n)	Rewire pointers

Search Algorithm:

```c
Node* search(SkipList* sl, int key) {
    Node* cur = sl->head;
    for (int lvl = sl->level; lvl >= 0; lvl--) {
        while (cur->forward[lvl] && cur->forward[lvl]->key < key)
            cur = cur->forward[lvl];
    }
    cur = cur->forward[0];
    if (cur && cur->key == key) return cur;
    return NULL;
}
```

Skip Lists are simple, fast, and probabilistically balanced , no rotations, no rebalancing.

D. Why Use Skip Lists?

- Easier to implement than balanced trees- Support concurrent access well- Randomized, not deterministic , but highly reliable Used in:
- Redis (sorted sets)- LevelDB / RocksDB internals- Concurrent maps

2. B-Trees

Invented by: Rudolf Bayer & Ed McCreight (1972) Goal: Reduce disk access by grouping data in blocks.

A B-Tree is a generalization of a BST:

- Each node holds multiple keys and children- Keys are kept sorted- Child subtrees span ranges between keys

A. Structure

A B-Tree of order `m`:

- Each node has `m` children- Each internal node has `k-1` keys if it has `k` children- All leaves at the same depth Example (order 3):

```
       [17 | 35]
      /    |     \
 [5 10] [20 25 30] [40 45 50]
```

B. Operations

1. Search
   - Traverse from root - Binary search in each node's key array - Follow appropriate child → $(O\log_m n)$
2. Insert
   - Insert in leaf - If overflow → split node - Promote median key to parent
3. Delete
   - Borrow or merge if node underflows Each split or merge keeps height minimal.

C. Complexity

Operation	Time	Disk Accesses	Notes
Search	$(O\log_m n)$	$(O\log_m n)$	m = branching factor
Insert	$(O\log_m n)$	(O(1)) splits	Balanced
Delete	$(O\log_m n)$	(O(1)) merges	Balanced

Height $\log_m n$ → very shallow when (m) large (e.g. 100).

D. B+ Tree Variant

In B+ Trees:

- All data in leaves (internal nodes = indexes)- Leaves linked → efficient range queries Used in:
- Databases (MySQL, PostgreSQL)- File systems (NTFS, HFS+)- Key-value stores

E. Example Flow

Insert 25:

```
[10 | 20 | 30] → overflow
Split → [10] [30]
Promote 20
Root: [20]
```

3. Comparison

Feature	Skip List	B-Tree
Balancing	Randomized	Deterministic
Fanout	2 (linked)	m-way
Environment	In-memory	Disk-based
Search	O(log n)	$O log_m n$
Insert/Delete	O(log n)	$O log_m n$
Concurrency	Easy	Complex
Range Queries	Sequential scan	Linked leaves (B+)

Tiny Code

Skip List Search (Conceptual):

```c
Node* search(SkipList* list, int key) {
    Node* cur = list->head;
    for (int lvl = list->level; lvl >= 0; lvl--) {
        while (cur->next[lvl] && cur->next[lvl]->key < key)
            cur = cur->next[lvl];
    }
    cur = cur->next[0];
    return (cur && cur->key == key) ? cur : NULL;
}
```

B-Tree Node (Skeleton):

```c
#define M 4
typedef struct {
    int keys[M-1];
    Node* child[M];
    int n;
} Node;
```

Why It Matters

Skip Lists and B-Trees show two paths to balance:

- Randomized simplicity (Skip List)- Block-based order (B-Tree) Both offer logarithmic guarantees, but one optimizes pointer chasing, the other I/O.

They're fundamental to:

- In-memory caches (Skip List)- On-disk indexes (B-Tree, B+ Tree)- Sorted data structures across systems

Try It Yourself

1. Build a basic skip list and insert random keys.
2. Trace a search path across levels.
3. Implement B-Tree insert and split logic.
4. Compare height of BST vs B-Tree for 1,000 keys.
5. Explore how Redis and MySQL use these internally.

Together, they form the bridge between linked lists and balanced trees, uniting speed, structure, and scalability.

29. Persistent and Functional Data Structures

Most data structures are ephemeral , when you update them, the old version disappears. But sometimes, you want to keep all past versions, so you can go back in time, undo operations, or run concurrent reads safely.

That's the magic of persistent data structures: every update creates a new version while sharing most of the old structure.

This section introduces the idea of persistence, explores how to make classic structures like arrays and trees persistent, and explains why functional programming loves them.

1. What Is Persistence?

A persistent data structure preserves previous versions after updates. You can access any version , past or present , without side effects.

Three levels:

Type	Description	Example
Partial	Can access past versions, but only modify the latest	Undo stack
Full	Can access and modify any version	Immutable map
Confluent	Can combine different versions	Git-like merges

This is essential in functional programming, undo systems, version control, persistent segment trees, and immutable databases.

2. Ephemeral vs Persistent

Ephemeral:

```
arr[2] = 7; // old value lost forever
```

Persistent:

```
new_arr = update(arr, 2, 7); // old_arr still exists
```

Persistent structures use structural sharing , unchanged parts are reused, not copied.

3. Persistent Linked List

Easiest example: each update creates a new head, reusing the tail.

```
struct Node { int val; Node* next; };

Node* push(Node* head, int x) {
    Node* newHead = malloc(sizeof(Node));
    newHead->val = x;
    newHead->next = head;
    return newHead;
}
```

Now both `old_head` and `new_head` coexist. Each version is immutable , you never change existing nodes.

Access: old and new lists share most of their structure:

```
v0: 1 → 2 → 3
v1: 0 → 1 → 2 → 3
```

Only one new node was created.

4. Persistent Binary Tree

For trees, updates create new paths from the root to the modified node, reusing the rest.

```
typedef struct Node {
    int key;
    struct Node *left, *right;
} Node;

Node* update(Node* root, int pos, int val) {
    if (!root) return newNode(val);
    Node* node = malloc(sizeof(Node));
    *node = *root; // copy
    if (pos < root->key) node->left = update(root->left, pos, val);
    else node->right = update(root->right, pos, val);
    return node;
}
```

Each `update` creates a new version , only $(O\log n)$ new nodes per change.

This is the core of persistent segment trees used in competitive programming.

5. Persistent Array (Functional Trick)

Arrays are trickier because of random access. Solutions:

- Use balanced binary trees as array replacements- Each update replaces one node- Persistent vector = tree of small arrays (used in Clojure, Scala) This gives:
- Access: $(O\log n)$- Update: $(O\log n)$- Space: $(O\log n)$ per update

6. Persistent Segment Tree

Used for versioned range queries:

- Each update = new root- Each version = snapshot of history Example: Track how array changes over time, query "sum in range [L,R] at version t".

Build:

```
Node* build(int L, int R) {
    if (L == R) return newNode(arr[L]);
    int mid = (L+R)/2;
    return newNode(
        build(L, mid),
        build(mid+1, R),
        sum
    );
}
```

Update: only $(O\log n)$ new nodes

```
Node* update(Node* prev, int L, int R, int pos, int val) {
    if (L == R) return newNode(val);
    int mid = (L+R)/2;
    if (pos <= mid)
        return newNode(update(prev->left, L, mid, pos, val), prev->right);
    else
        return newNode(prev->left, update(prev->right, mid+1, R, pos, val));
}
```

Each version = new root; old ones still valid.

7. Functional Perspective

In functional programming, data is immutable by default. Instead of mutating, you create a new version.

This allows:

- Thread-safety (no races)- Time-travel debugging- Undo/redo systems- Concurrency without locks Languages like Haskell, Clojure, and Elm build everything this way.

For example, Clojure's `persistent vector` uses path copying and branching factor 32 for $(O\log_{32} n)$ access.

8. Applications

- Undo / Redo stacks (text editors, IDEs)- Version control (Git trees)- Immutable databases (Datomic)- Segment trees over time (competitive programming)- Snapshots in memory allocators or games

9. Complexity

Structure	Update	Access	Space per Update	Notes
Persistent Linked List	O(1)	O(1)	O(1)	Simple sharing
Persistent Tree	O(log n)	O(log n)	O(log n)	Path copying
Persistent Array	O(log n)	O(log n)	O(log n)	Tree-backed
Persistent Segment Tree	O(log n)	O(log n)	O(log n)	Versioned queries

Tiny Code

Persistent Linked List Example:

```
Node* v0 = NULL;
v0 = push(v0, 3);
v0 = push(v0, 2);
Node* v1 = push(v0, 1);
// v0 = [2,3], v1 = [1,2,3]
```

Why It Matters

Persistence is about time as a first-class citizen. It lets you:

- Roll back- Compare versions- Work immutably and safely It's the algorithmic foundation behind functional programming, time-travel debugging, and immutable data systems.

It teaches this powerful idea:

> "Never destroy , always build upon what was."

Try It Yourself

1. Implement a persistent stack using linked lists.
2. Write a persistent segment tree for range sums.
3. Track array versions after each update and query old states.
4. Compare space/time with an ephemeral one.
5. Explore persistent structures in Clojure (`conj`, `assoc`) or Rust (`im` crate).

Persistence transforms data from fleeting state into a history you can navigate , a timeline of structure and meaning.

30. Advanced Trees and Range Queries

So far, you've seen balanced trees (AVL, Red-Black, Treap) and segment-based structures (Segment Trees, Fenwick Trees). Now it's time to combine those ideas and step into advanced trees , data structures that handle dynamic sets, order statistics, intervals, ranges, and geometry-like queries in logarithmic time.

This chapter is about trees that go beyond search , they store order, track ranges, and answer complex queries efficiently.

We'll explore:

- Order Statistic Trees (k-th element, rank queries)- Interval Trees (range overlaps)- Range Trees (multi-dimensional search)- KD-Trees (spatial partitioning)- Merge Sort Trees (offline range queries)

1. Order Statistic Tree

Goal: find the k-th smallest element, or the rank of an element, in $(O \log n)$.

Built on top of a balanced BST (e.g. Red-Black) by storing subtree sizes.

A. Augmented Tree Nodes

Each node keeps:

- `key`: element value- `left`, `right`: children- `size`: number of nodes in subtree

```
typedef struct Node {
    int key, size;
    struct Node *left, *right;
} Node;
```

Whenever you rotate or insert, update `size`:

```
int get_size(Node* n) { return n ? n->size : 0; }
void update_size(Node* n) {
    if (n) n->size = get_size(n->left) + get_size(n->right) + 1;
}
```

B. Find k-th Element

Recursively use subtree sizes:

```
Node* kth(Node* root, int k) {
    int left = get_size(root->left);
    if (k == left + 1) return root;
    else if (k <= left) return kth(root->left, k);
    else return kth(root->right, k - left - 1);
}
```

Time: $(O \log n)$

C. Find Rank

Find position of a key (number of smaller elements):

```
int rank(Node* root, int key) {
    if (!root) return 0;
    if (key < root->key) return rank(root->left, key);
    if (key > root->key) return get_size(root->left) + 1 + rank(root->right, key);
    return get_size(root->left) + 1;
}
```

Used in:

- Databases (ORDER BY, pagination)- Quantile queries- Online median maintenance

2. Interval Tree

Goal: find all intervals overlapping with a given point or range.

Used in computational geometry, scheduling, and genomic data.

A. Structure

BST ordered by interval low endpoint. Each node stores:

- `low`, `high`: interval bounds- `max`: maximum `high` in its subtree

```
typedef struct {
    int low, high, max;
    struct Node *left, *right;
} Node;
```

B. Query Overlap

Check if `x` overlaps `node->interval`: If not, go left or right based on `max` values.

```
bool overlap(Interval a, Interval b) {
    return a.low <= b.high && b.low <= a.high;
}

Node* overlap_search(Node* root, Interval q) {
    if (!root) return NULL;
    if (overlap(root->interval, q)) return root;
    if (root->left && root->left->max >= q.low)
        return overlap_search(root->left, q);
    return overlap_search(root->right, q);
}
```

Time: $(O\log n)$ average

C. Use Cases

- Calendar/schedule conflict detection- Collision detection- Genome region lookup- Segment intersection

3. Range Tree

Goal: multi-dimensional queries like

> "How many points fall inside rectangle [x1, x2] $\times$ [y1, y2]?"

Structure:

- Primary BST on x- Each node stores secondary BST on y Query time: $(O\log^2 n)$ Space: $(On\log n)$

Used in:

- 2D search- Computational geometry- Databases (spatial joins)

4. KD-Tree

Goal: efficiently search points in k-dimensional space.

Alternate splitting dimensions at each level:

- Level 0 → split by x- Level 1 → split by y- Level 2 → split by z Each node stores:
- Point (vector)- Split axis Used for:
- Nearest neighbor search- Range queries- ML (k-NN classifiers) Time:
- Build: $(On\log n)$- Query: $(O\sqrt{n})$ average in 2D

5. Merge Sort Tree

Goal: query "number of elements   k in range [L, R]"

Built like a segment tree, but each node stores a sorted list of its range.

Build: merge children lists Query: binary search in node lists

Time:

- Build: $(On\log n)$- Query: $(O\log^2 n)$ Used in offline queries and order-statistics over ranges.

6. Comparison

Tree Type	Use Case	Query	Update	Notes
Order Statistic	k-th, rank	O(log n)	O(log n)	Augmented BST
Interval	Overlaps	O(log n + k)	O(log n)	Store intervals
Range Tree	2D range	O($\log^2$ n + k)	O($\log^2$ n)	Multi-dim
KD-Tree	Spatial	O(√n) avg	O(log n)	Nearest neighbor
Merge Sort Tree	Offline rank	O($\log^2$ n)	Static	Built from sorted segments

Tiny Code

Order Statistic Example:

```
Node* root = NULL;
root = insert(root, 10);
root = insert(root, 20);
root = insert(root, 30);
printf("%d", kth(root, 2)->key); // 20
```

Interval Query:

```
Interval q = {15, 17};
Node* res = overlap_search(root, q);
if (res) printf("Overlap: [%d, %d]\n", res->low, res->high);
```

Why It Matters

These trees extend balance into dimensions and ranges. They let you query ordered data efficiently: "How many?", "Which overlaps?", "Where is k-th smallest?".

They teach a deeper design principle:

> "Augment structure with knowledge , balance plus metadata equals power."

Try It Yourself

1. Implement an order statistic tree , test rank/k-th queries.
2. Insert intervals and test overlap detection.
3. Build a simple KD-tree for 2D points.
4. Solve rectangle counting with a range tree.
5. Precompute a merge sort tree for offline queries.

These advanced trees form the final evolution of structured queries , blending geometry, order, and logarithmic precision.

Chapter 4. Graph Algorithms

31. Traversals (DFS, BFS, Iterative Deepening)

Graphs are everywhere , maps, networks, dependencies, state spaces. Before you can analyze them, you need a way to visit their vertices , systematically, without getting lost or looping forever.

That's where graph traversals come in. They're the foundation for everything that follows: connected components, shortest paths, spanning trees, topological sorts, and more.

This section walks through the three pillars:

- DFS (Depth-First Search) , explore deeply before backtracking- BFS (Breadth-First Search) , explore level by level- Iterative Deepening , a memory-friendly hybrid

1. Representing Graphs

Before traversal, you need a good structure.

Adjacency List (most common):

```
#define MAX 1000
vector<int> adj[MAX];
```

Add edges:

```
void add_edge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u); // omit if directed
}
```

Track visited vertices:

```
bool visited[MAX];
```

2. Depth-First Search (DFS)

DFS dives deep, following one branch fully before exploring others. It's recursive, like exploring a maze by always turning left until you hit a wall.

A. Recursive Form

```
void dfs(int u) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v])
            dfs(v);
    }
}
```

Start it:

```
dfs(start_node);
```

B. Iterative Form (with Stack)

```
void dfs_iter(int start) {
    stack<int> s;
    s.push(start);
    while (!s.empty()) {
        int u = s.top(); s.pop();
        if (visited[u]) continue;
        visited[u] = true;
        for (int v : adj[u]) s.push(v);
    }
}
```

C. Complexity

Graph Type	Time	Space
Adjacency List	O(V + E)	O(V)

DFS is used in:

- Connected components- Cycle detection- Topological sort- Backtracking & search- Articulation points / bridges

3. Breadth-First Search (BFS)

BFS explores neighbors first , it's like expanding in waves. This guarantees shortest path in unweighted graphs.

A. BFS with Queue

```cpp
void bfs(int start) {
    queue<int> q;
    q.push(start);
    visited[start] = true;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}
```

B. Track Distance

```cpp
int dist[MAX];
void bfs_dist(int s) {
    fill(dist, dist + MAX, -1);
    dist[s] = 0;
    queue<int> q; q.push(s);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) {
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
}
```

Now `dist[v]` gives shortest distance from `s`.

C. Complexity

Same as DFS:

Time	Space
O(V + E)	O(V)

Used in:

- Shortest paths (unweighted)- Level-order traversal- Bipartite check- Connected components

4. Iterative Deepening Search (IDS)

DFS is memory-light but might go too deep. BFS is optimal but can use a lot of memory. Iterative Deepening Search (IDS) combines both.

It performs DFS with increasing depth limits:

```
bool dls(int u, int target, int depth) {
    if (u == target) return true;
    if (depth == 0) return false;
    for (int v : adj[u])
        if (dls(v, target, depth - 1)) return true;
    return false;
}

bool ids(int start, int target, int max_depth) {
    for (int d = 0; d <= max_depth; d++)
        if (dls(start, target, d)) return true;
    return false;
}
```

Used in:

- AI search problems (state spaces)- Game trees (chess, puzzles)

5. Traversal Order Examples

For a graph:

```
1 - 2 - 3
|   |
4 - 5
```

DFS (starting at 1): 1 → 2 → 3 → 5 → 4 BFS (starting at 1): 1 → 2 → 4 → 3 → 5

6. Directed vs Undirected

- Undirected: mark both directions- Directed: follow edge direction only DFS on directed graphs is core to:
- SCC (Strongly Connected Components)- Topological Sorting- Reachability analysis

7. Traversal Trees

Each traversal implicitly builds a spanning tree:

- DFS Tree: based on recursion- BFS Tree: based on levels Use them to:
- Detect cross edges, back edges- Classify edges (important for algorithms like Tarjan's)

8. Comparison

Aspect	DFS	BFS
Strategy	Deep first	Level-wise
Space	O(V) (stack)	O(V) (queue)
Path Optimality	Not guaranteed	Yes (unweighted)
Applications	Cycle detection, backtracking	Shortest path, level order

Tiny Code

DFS + BFS Combo:

```
void traverse(int n) {
    for (int i = 1; i <= n; i++) visited[i] = false;
    dfs(1);
    for (int i = 1; i <= n; i++) visited[i] = false;
    bfs(1);
}
```

Why It Matters

DFS and BFS are the roots of graph theory in practice. Every algorithm you'll meet later , shortest paths, flows, SCCs , builds upon them.

They teach you how to navigate structure, how to systematically explore unknowns, and how search lies at the heart of computation.

Try It Yourself

1. Build an adjacency list and run DFS/BFS from vertex 1.
2. Track discovery and finish times in DFS.
3. Use BFS to compute shortest paths in an unweighted graph.
4. Modify DFS to count connected components.
5. Implement IDS for a puzzle like the 8-puzzle.

Graph traversal is the art of exploration , once you master it, the rest of graph theory falls into place.

32. Strongly Connected Components (Tarjan, Kosaraju)

In directed graphs, edges have direction, so connectivity gets tricky. It's not enough for vertices to be reachable , you need mutual reachability.

That's the essence of a strongly connected component (SCC):

> A set of vertices where every vertex can reach every other vertex.

Think of SCCs as islands of mutual connectivity , inside, you can go anywhere; outside, you can't. They're the building blocks for simplifying directed graphs into condensation DAGs (no cycles).

We'll explore two classic algorithms:

- Kosaraju's Algorithm , clean, intuitive, two-pass- Tarjan's Algorithm , one-pass, stack-based elegance

1. Definition

A Strongly Connected Component (SCC) in a directed graph (G = (V, E)) is a maximal subset of vertices $C \subseteq V$ such that for every pair ((u, v) C): $u \to v$ and $v \to u$.

In other words, every node in (C) is reachable from every other node in (C).

Example:

```
1 → 2 → 3 → 1   forms an SCC
4 → 5           separate SCCs
```

2. Applications

- Condensation DAG: compress SCCs into single nodes , no cycles remain.- Component-based reasoning: topological sorting on DAG of SCCs.- Program analysis: detecting cycles, dependencies.- Web graphs: find clusters of mutually linked pages.- Control-flow: loops and strongly connected subroutines.

3. Kosaraju's Algorithm

A simple two-pass algorithm using DFS and graph reversal.

Steps:

1. Run DFS and push nodes onto a stack in finish-time order.
2. Reverse the graph (edges flipped).
3. Pop nodes from stack; DFS on reversed graph; each DFS = one SCC.

A. Implementation

```
vector<int> adj[MAX], rev[MAX];
bool visited[MAX];
stack<int> st;
vector<vector<int>> sccs;

void dfs1(int u) {
    visited[u] = true;
    for (int v : adj[u])
        if (!visited[v])
            dfs1(v);
    st.push(u);
}

void dfs2(int u, vector<int>& comp) {
    visited[u] = true;
    comp.push_back(u);
    for (int v : rev[u])
        if (!visited[v])
            dfs2(v, comp);
}

void kosaraju(int n) {
    // Pass 1: order by finish time
    for (int i = 1; i <= n; i++)
        if (!visited[i]) dfs1(i);

    // Reverse graph
    for (int u = 1; u <= n; u++)
        for (int v : adj[u])
            rev[v].push_back(u);

    // Pass 2: collect SCCs
    fill(visited, visited + n + 1, false);
    while (!st.empty()) {
        int u = st.top(); st.pop();
        if (!visited[u]) {
            vector<int> comp;
            dfs2(u, comp);
            sccs.push_back(comp);
        }
    }
}
```

Time Complexity: (O(V + E)) , two DFS passes.

Space Complexity: (O(V + E))

B. Example

Graph:

```
1 → 2 → 3
↑   ↓   ↓
5 ← 4 ← 6
```

SCCs:

- {1,2,4,5}- {3,6}

4. Tarjan's Algorithm

More elegant: one DFS pass, no reversal, stack-based. It uses discovery times and low-link values to detect SCC roots.

A. Idea

- `disc[u]`: discovery time of node `u`- `low[u]`: smallest discovery time reachable from `u`- A node is root of an SCC if `disc[u] == low[u]` Maintain a stack of active nodes (in current DFS path).

B. Implementation

```
vector<int> adj[MAX];
int disc[MAX], low[MAX], timer;
bool inStack[MAX];
stack<int> st;
vector<vector<int>> sccs;

void dfs_tarjan(int u) {
    disc[u] = low[u] = ++timer;
    st.push(u);
    inStack[u] = true;

    for (int v : adj[u]) {
        if (!disc[v]) {
            dfs_tarjan(v);
            low[u] = min(low[u], low[v]);
        } else if (inStack[v]) {
            low[u] = min(low[u], disc[v]);
        }
    }

    if (disc[u] == low[u]) {
        vector<int> comp;
        while (true) {
            int v = st.top(); st.pop();
            inStack[v] = false;
            comp.push_back(v);
            if (v == u) break;
        }
        sccs.push_back(comp);
    }
}

void tarjan(int n) {
    for (int i = 1; i <= n; i++)
        if (!disc[i])
            dfs_tarjan(i);
}
```

Time Complexity: (O(V + E))

Space Complexity: (O(V))

C. Walkthrough

Graph:

```
1 → 2 → 3
↑   ↓   ↓
5 ← 4 ← 6
```

DFS visits nodes in order; when it finds a node whose `disc == low`, it pops from the stack to form an SCC.

Result:

```
SCC1: 1 2 4 5
SCC2: 3 6
```

5. Comparison

Feature	Kosaraju	Tarjan
DFS Passes	2	1
Reversal Needed	Yes	No
Stack	Yes (finish order)	Yes (active path)
Implementation	Simple conceptually	Compact, efficient
Time	O(V + E)	O(V + E)

6. Condensation Graph

Once SCCs are found, you can build a DAG: Each SCC becomes a node, edges represent cross-SCC connections. Topological sorting now applies.

Used in:

- Dependency analysis- Strong component compression- DAG dynamic programming

Tiny Code

Print SCCs (Tarjan):

```
tarjan(n);
for (auto &comp : sccs) {
    for (int x : comp) printf("%d ", x);
    printf("\n");
}
```

Why It Matters

SCC algorithms turn chaotic directed graphs into structured DAGs. They're the key to reasoning about cycles, dependencies, and modularity.

Understanding them reveals a powerful truth:

> "Every complex graph can be reduced to a simple hierarchy , once you find its strongly connected core."

Try It Yourself

1. Implement both Kosaraju and Tarjan , verify they match.
2. Build SCC DAG and run topological sort on it.
3. Detect cycles via SCC size > 1.
4. Use SCCs to solve 2-SAT (boolean satisfiability).
5. Visualize condensation of a graph with 6 nodes.

Once you can find SCCs, you can tame directionality , transforming messy networks into ordered systems.

33. Shortest Paths (Dijkstra, Bellman-Ford, A*, Johnson)

Once you can traverse a graph, the next natural question is:

> "What is the shortest path between two vertices?"

Shortest path algorithms are the heart of routing, navigation, planning, and optimization. They compute minimal cost paths , whether distance, time, or weight , and adapt to different edge conditions (non-negative, negative, heuristic).

This section covers the most essential algorithms:

- Dijkstra's Algorithm , efficient for non-negative weights- Bellman-Ford Algorithm , handles negative edges- A* , best-first with heuristics- Johnson's Algorithm , all-pairs shortest paths in sparse graphs

1. The Shortest Path Problem

Given a weighted graph (G = (V, E)) and a source (s), find dist[v], the minimum total weight to reach every vertex (v).

Variants:

- Single-source shortest path (SSSP) , one source to all- Single-pair , one source to one target- All-pairs shortest path (APSP) , every pair- Dynamic shortest path , with updates

2. Dijkstra's Algorithm

Best for non-negative weights. Idea: explore vertices in increasing distance order, like water spreading.

A. Steps

1. Initialize all distances to infinity.
2. Set source distance = 0.
3. Use a priority queue to always pick the node with smallest tentative distance.
4. Relax all outgoing edges.

B. Implementation (Adjacency List)

```cpp
#include <bits/stdc++.h>
using namespace std;

const int INF = 1e9;
vector<pair<int,int>> adj[1000]; // (neighbor, weight)
int dist[1000];

void dijkstra(int n, int s) {
    fill(dist, dist + n + 1, INF);
    dist[s] = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
    pq.push({0, s});

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d != dist[u]) continue;
        for (auto [v, w] : adj[u]) {
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
}
```

Complexity:

- Using priority queue (binary heap): $O((V + E)\log V)$
- Space: $O(V + E)$

C. Example

Graph:

```
1 →(2) 2 →(3) 3
↓(4)       ↑(1)
4 →(2)
```

`dijkstra(1)` gives shortest distances:

```
dist[1] = 0
dist[2] = 2
dist[3] = 5
dist[4] = 4
```

D. Properties

- Works only if all edges $w \geq 0$- Can reconstruct path via `parent[v]`- Used in:
  - GPS and routing systems - Network optimization - Scheduling with positive costs

3. Bellman-Ford Algorithm

Handles negative edge weights, and detects negative cycles.

A. Idea

Relax all edges (V-1) times. If on (V)-th iteration you can still relax → negative cycle exists.

B. Implementation

```
struct Edge { int u, v, w; };
vector<Edge> edges;
int dist[1000];

bool bellman_ford(int n, int s) {
    fill(dist, dist + n + 1, INF);
    dist[s] = 0;
    for (int i = 1; i <= n - 1; i++) {
        for (auto e : edges) {
            if (dist[e.u] + e.w < dist[e.v])
                dist[e.v] = dist[e.u] + e.w;
        }
    }
    // Check for negative cycle
    for (auto e : edges)
        if (dist[e.u] + e.w < dist[e.v])
            return false; // negative cycle
    return true;
}
```

Complexity: (O(VE)) Works even when (w < 0).

C. Example

Graph:

```
1 →(2) 2 →(-5) 3 →(2) 4
```

Bellman-Ford finds path 1→2→3→4 with total cost (-1).

If a cycle reduces total weight indefinitely, algorithm detects it.

D. Use Cases

- Currency exchange arbitrage- Game graphs with penalties- Detecting impossible constraints

4. A* Search Algorithm

Heuristic-guided shortest path, perfect for pathfinding (AI, maps, games).

It combines actual cost and estimated cost:

$$f(v) = g(v) + h(v)$$

where

- (g(v)): known cost so far- (h(v)): heuristic estimate (must be admissible)

A. Pseudocode

```
priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
g[start] = 0;
pq.push({h[start], start});

while (!pq.empty()) {
    auto [f, u] = pq.top(); pq.pop();
    if (u == goal) break;
    for (auto [v, w] : adj[u]) {
        int new_g = g[u] + w;
        if (new_g < g[v]) {
            g[v] = new_g;
            pq.push({g[v] + h[v], v});
        }
    }
}
```

Heuristic Example:

- Euclidean distance (for grids)- Manhattan distance (for 4-direction movement)

B. Use Cases

- Game AI (pathfinding)- Robot motion planning- Map navigation Complexity: (O(E)) in best case, depends on heuristic quality.

5. Johnson's Algorithm

Goal: All-Pairs Shortest Path in sparse graphs with negative edges (no negative cycles).

Idea:

1. Add new vertex `q` connected to all others with edge weight 0
2. Run Bellman-Ford from `q` to get potential `h(v)`
3. Reweight edges: (w'(u, v) = w(u, v) + h(u) - h(v)) (now all weights 0)
4. Run Dijkstra from each vertex

Complexity: $(\mathrm{O}VE + V^2 \log V)$

6. Summary

Algorithm	Handles Negative Weights	Detects Negative Cycle	Heuristic	Complexity	Use Case
Dijkstra	No	No	No	O((V+E) log V)	Non-negative weights
Bellman-Ford	Yes	Yes	No	O(VE)	Negative edges
A*	No (unless careful)	No	Yes	Depends	Pathfinding
Johnson	Yes (no neg. cycles)	Yes	No	O(VE + V log V)	All-pairs, sparse

Tiny Code

Dijkstra Example:

```
dijkstra(n, 1);
for (int i = 1; i <= n; i++)
    printf("dist[%d] = %d\n", i, dist[i]);
```

Why It Matters

Shortest paths are the essence of optimization , not just in graphs, but in reasoning: finding minimal cost, minimal distance, minimal risk.

These algorithms teach:

> "The path to a goal isn't random , it's guided by structure, weight, and knowledge."

Try It Yourself

1. Build a weighted graph and compare Dijkstra vs Bellman-Ford.
2. Introduce a negative edge and observe Bellman-Ford detecting it.
3. Implement A* on a grid with obstacles.
4. Use Dijkstra to plan routes in a city map dataset.
5. Try Johnson's algorithm for all-pairs shortest paths.

Master these, and you master direction + cost = intelligence in motion.

34. Shortest Path Variants (0-1 BFS, Bidirectional, Heuristic A*)

Sometimes the classic shortest path algorithms aren't enough. You might have special edge weights (only 0 or 1), a need for faster searches, or extra structure you can exploit.

That's where shortest path variants come in , they're optimized adaptations of the big three (BFS, Dijkstra, A*) for specific scenarios.

In this section, we'll explore:

- 0-1 BFS → when edge weights are only 0 or 1- Bidirectional Search → meet-in-the-middle for speed- Heuristic A* → smarter exploration guided by estimates Each shows how structure in your problem can yield speed-ups.

1. 0-1 BFS

If all edge weights are either 0 or 1, you don't need a priority queue. A deque (double-ended queue) is enough for (O(V + E)) time.

Why? Because edges with weight 0 should be processed immediately, while edges with weight 1 can wait one step longer.

A. Algorithm

Use a deque.

- When relaxing an edge with weight 0, push to front.- When relaxing an edge with weight 1, push to back.

```cpp
const int INF = 1e9;
vector<pair<int,int>> adj[1000]; // (v, w)
int dist[1000];

void zero_one_bfs(int n, int s) {
    fill(dist, dist + n + 1, INF);
    deque<int> dq;
    dist[s] = 0;
    dq.push_front(s);

    while (!dq.empty()) {
        int u = dq.front(); dq.pop_front();
        for (auto [v, w] : adj[u]) {
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                if (w == 0) dq.push_front(v);
                else dq.push_back(v);
            }
        }
    }
}
```

B. Example

Graph:

```
1 -0-> 2 -1-> 3
|             ^
1             |
+-------------+
```

Shortest path from 1 to 3 = 1 (via edge 1-2-3). Deque ensures weight-0 edges don't get delayed.

C. Complexity

Time	Space	Notes
O(V + E)	O(V)	Optimal for binary weights

Used in:

- Layered BFS- Grid problems with binary costs- BFS with teleportation (weight 0 edges)

2. Bidirectional Search

Sometimes you just need one path , from source to target , in an unweighted graph. Instead of expanding from one side, expand from both ends and stop when they meet.

This reduces search depth from (Ob^d) to (O$b^{d/2}$) (huge gain for large graphs).

A. Idea

Run BFS from both source and target simultaneously. When their frontiers intersect, you've found the shortest path.

B. Implementation

```
bool visited_from_s[MAX], visited_from_t[MAX];
queue<int> qs, qt;

int bidirectional_bfs(int s, int t) {
    qs.push(s); visited_from_s[s] = true;
    qt.push(t); visited_from_t[t] = true;

    while (!qs.empty() && !qt.empty()) {
        if (step(qs, visited_from_s, visited_from_t)) return 1;
        if (step(qt, visited_from_t, visited_from_s)) return 1;
    }
    return 0;
}

bool step(queue<int>& q, bool vis[], bool other[]) {
    int size = q.size();
    while (size--) {
        int u = q.front(); q.pop();
        if (other[u]) return true;
        for (int v : adj[u]) {
            if (!vis[v]) {
                vis[v] = true;
                q.push(v);
            }
        }
    }
    return false;
}
```

C. Complexity

Time	Space	Notes
O$b^{d/2}$	O$b^{d/2}$	Doubly fast in practice

Used in:

- Maze solvers- Shortest paths in large sparse graphs- Social network "degrees of separation"

3. Heuristic A* (Revisited)

A* generalizes Dijkstra with goal-directed search using heuristics. We revisit it here to show how heuristics change exploration order.

A. Cost Function

$$f(v) = g(v) + h(v)$$

- (g(v)): cost so far- (h(v)): estimated cost to goal- (h(v)) must be admissible ((h(v) true cost))

B. Implementation

```
struct Node {
    int v; int f, g;
    bool operator>(const Node& o) const { return f > o.f; }
};

priority_queue<Node, vector<Node>, greater<Node>> pq;

void astar(int start, int goal) {
    g[start] = 0;
    h[start] = heuristic(start, goal);
    pq.push({start, g[start] + h[start], g[start]});

    while (!pq.empty()) {
        auto [u, f_u, g_u] = pq.top(); pq.pop();
        if (u == goal) break;
        for (auto [v, w] : adj[u]) {
            int new_g = g[u] + w;
            if (new_g < g[v]) {
                g[v] = new_g;
                int f_v = new_g + heuristic(v, goal);
                pq.push({v, f_v, new_g});
            }
        }
    }
}
```

C. Example Heuristics

- Grid map: Manhattan distance (h(x, y) = |x - x_g| + |y - y_g|)
- Navigation: straight-line (Euclidean)- Game tree: evaluation function

D. Performance

Heuristic	Effect
Perfect (h = true cost)	Optimal, visits minimal nodes
Admissible but weak	Still correct, more nodes
Overestimate	May fail (non-admissible)

4. Comparison

Algorithm	Weight Type	Strategy	Time	Space	Notes
0-1 BFS	0 or 1	Deque-based	O(V+E)	O(V)	No heap
Bidirectional BFS	Unweighted	Two-way search	O$b^{d/2}$	O$b^{d/2}$	Meets in middle
A*	Non-negative	Heuristic search	Depends	O(V)	Guided

5. Example Scenario

Problem	Variant
Grid with teleport (cost 0)	0-1 BFS
Huge social graph (find shortest chain)	Bidirectional BFS
Game AI pathfinding	A* with Manhattan heuristic

Tiny Code

0-1 BFS Quick Demo:

```
add_edge(1, 2, 0);
add_edge(2, 3, 1);
zero_one_bfs(3, 1);
printf("%d\n", dist[3]); // shortest = 1
```

Why It Matters

Special cases deserve special tools. These variants show that understanding structure (like edge weights or symmetry) can yield huge gains.

They embody a principle:

> "Don't just run faster , run smarter, guided by what you know."

Try It Yourself

1. Implement 0-1 BFS for a grid with cost 0 teleports.
2. Compare BFS vs Bidirectional BFS on a large maze.
3. Write A* for an 8x8 chessboard knight's move puzzle.
4. Tune heuristics , see how overestimating breaks A*.
5. Combine A* and 0-1 BFS for hybrid search.

With these in hand, you can bend shortest path search to the shape of your problem , efficient, elegant, and exact.

35. Minimum Spanning Trees (Kruskal, Prim, Borůvka)

When a graph connects multiple points with weighted edges, sometimes you don't want the *shortest path*, but the *cheapest network* that connects everything.

That's the Minimum Spanning Tree (MST) problem:

> Given a connected, weighted, undirected graph, find a subset of edges that connects all vertices with minimum total weight and no cycles.

MSTs are everywhere , from building networks and designing circuits to clustering and approximation algorithms.

Three cornerstone algorithms solve it beautifully:

- Kruskal’s , edge-based, union-find- Prim’s , vertex-based, greedy expansion- Borůvka’s , component merging in parallel

1. What Is a Spanning Tree?

A spanning tree connects all vertices with exactly (V-1) edges. Among all spanning trees, the one with minimum total weight is the MST.

Properties:

- Contains no cycles- Connects all vertices- Edge count = (V - 1)- Unique if all weights distinct

2. MST Applications

- Network design (roads, cables, pipelines)- Clustering (e.g., hierarchical clustering)- Image segmentation- Approximation (e.g., TSP ~ 2 × MST)- Graph simplification

3. Kruskal’s Algorithm

Build the MST edge-by-edge, in order of increasing weight. Use Union-Find (Disjoint Set Union) to avoid cycles.

A. Steps

1. Sort all edges by weight.
2. Initialize each vertex as its own component.
3. For each edge (u, v):
   - If `u` and `v` are in different components → include edge - Union their sets Stop when (V-1) edges chosen.

B. Implementation

```cpp
struct Edge { int u, v, w; };
vector<Edge> edges;
int parent[MAX], rank_[MAX];

int find(int x) {
    return parent[x] == x ? x : parent[x] = find(parent[x]);
}
bool unite(int a, int b) {
    a = find(a); b = find(b);
    if (a == b) return false;
    if (rank_[a] < rank_[b]) swap(a, b);
    parent[b] = a;
    if (rank_[a] == rank_[b]) rank_[a]++;
    return true;
}

int kruskal(int n) {
    iota(parent, parent + n + 1, 0);
    sort(edges.begin(), edges.end(), [](Edge a, Edge b){ return a.w < b.w; });
    int total = 0;
    for (auto &e : edges)
        if (unite(e.u, e.v))
            total += e.w;
    return total;
}
```

Complexity:

- Sorting edges: (O$E \log E$)- Union-Find operations: (O((V))) (almost constant)- Total: (O$E \log E$)

C. Example

Graph:

```
1 -4- 2
|     |
2     3
 \-1-/
```

Edges sorted: (1-3,1), (1-2,4), (2-3,3)

Pick 1-3, 2-3 $\rightarrow$ MST weight $= 1 + 3 = 4$

4. Prim's Algorithm

Grow MST from a starting vertex, adding the smallest outgoing edge each step.

Similar to Dijkstra , but pick edges, not distances.

A. Steps

1. Start with one vertex, mark as visited.
2. Use priority queue for candidate edges.
3. Pick smallest edge that connects to an unvisited vertex.
4. Add vertex to MST, repeat until all visited.

B. Implementation

```cpp
vector<pair<int,int>> adj[MAX]; // (v, w)
bool used[MAX];
int prim(int n, int start) {
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
    pq.push({0, start});
    int total = 0;

    while (!pq.empty()) {
        auto [w, u] = pq.top(); pq.pop();
        if (used[u]) continue;
        used[u] = true;
        total += w;
        for (auto [v, w2] : adj[u])
            if (!used[v]) pq.push({w2, v});
    }
    return total;
}
```

Complexity:

- $O((V + E)\log V)$ with binary heap

Used when:

- Graph is dense
- Easier to grow tree than sort all edges

C. Example

Graph:

```
1 -2- 2
|     |
4     1
 \-3-/
```

Start at 1 → choose (1-2), (1-3) → MST weight = 2 + 3 = 5

5. Borůvka's Algorithm

Less famous, but elegant , merges cheapest outgoing edge per component in parallel.

Each component picks one cheapest outgoing edge, adds it, merges components. Repeat until one component left.

Complexity: $(\mathrm{O}E \log V)$

Used in parallel/distributed MST computations.

6. Comparison

Algorithm	Strategy	Time	Space	Best For
Kruskal	Edge-based, sort all edges	O(E log E)	O(E)	Sparse graphs
Prim	Vertex-based, grow tree	O(E log V)	O(V+E)	Dense graphs
Borůvka	Component merging	O(E log V)	O(E)	Parallel MST

7. MST Properties

- Cut Property: For any cut, smallest crossing edge MST.- Cycle Property: For any cycle, largest edge not MST.- MST may not be unique if equal weights.

8. Building the Tree

Store MST edges:

```
vector<Edge> mst_edges;
if (unite(e.u, e.v)) mst_edges.push_back(e);
```

Then use MST for:

- Path queries- Clustering (remove largest edge)- Approximation TSP (preorder traversal)

Tiny Code

Kruskal Example:

```
edges.push_back({1,2,4});
edges.push_back({1,3,1});
edges.push_back({2,3,3});
printf("MST = %d\n", kruskal(3)); // 4
```

Why It Matters

MSTs model connection without redundancy. They're about efficiency , connecting everything at minimal cost, a principle that appears in infrastructure, data, and even ideas.

They teach:

> "You can connect the whole with less , if you choose wisely."

Try It Yourself

1. Implement Kruskal's algorithm using union-find.
2. Run Prim's algorithm and compare output.
3. Build MST on random weighted graph , visualize tree.
4. Remove heaviest edge from MST to form two clusters.
5. Explore Borůvka for parallel execution.

MSTs are how you span complexity with minimal effort , a tree of balance, economy, and order.

36. Flows (Ford-Fulkerson, Edmonds-Karp, Dinic)

Some graphs don't just connect , they *carry* something. Imagine water flowing through pipes, traffic through roads, data through a network. Each edge has a capacity, and you want to know:

> "How much can I send from source to sink before the system clogs?"

That's the Maximum Flow problem , a cornerstone of combinatorial optimization, powering algorithms for matching, cuts, scheduling, and more.

This section covers the big three:

- Ford-Fulkerson , the primal idea- Edmonds-Karp , BFS-based implementation- Dinic's Algorithm , layered speed

1. Problem Definition

Given a directed graph (G = (V, E)), each edge ((u, v)) has a capacity (c(u, v) 0).

We have:

- Source (s)- Sink (t) We want the maximum flow from (s) to (t): a function (f(u, v)) that satisfies:

1. Capacity constraint: (0 f(u, v) c(u, v))
2. Flow conservation: For every vertex $v \neq s, t$: (f(u, v) = f(v, w))

Total flow = (f(s, v))

2. The Big Picture

Max Flow - Min Cut Theorem:

> The value of the maximum flow equals the capacity of the minimum cut.

So finding a max flow is equivalent to finding the bottleneck.

3. Ford-Fulkerson Method

The idea:

- While there exists a path from (s) to (t) with available capacity, push flow along it.

Each step:

1. Find augmenting path
2. Send flow = min residual capacity along it
3. Update residual capacities

Repeat until no augmenting path.

A. Residual Graph

Residual capacity:

$$r(u, v) = c(u, v) - f(u, v)$$

If (f(u, v) > 0), then add reverse edge ((v, u)) with capacity (f(u, v)).

This allows undoing flow if needed.

B. Implementation (DFS-style)

```cpp
const int INF = 1e9;
vector<pair<int,int>> adj[MAX];
int cap[MAX][MAX];

int dfs(int u, int t, int flow, vector<int>& vis) {
    if (u == t) return flow;
    vis[u] = 1;
    for (auto [v, _] : adj[u]) {
        if (!vis[v] && cap[u][v] > 0) {
            int pushed = dfs(v, t, min(flow, cap[u][v]), vis);
            if (pushed > 0) {
                cap[u][v] -= pushed;
                cap[v][u] += pushed;
                return pushed;
            }
        }
    }
    return 0;
}

int ford_fulkerson(int s, int t, int n) {
    int flow = 0;
    while (true) {
        vector<int> vis(n + 1, 0);
        int pushed = dfs(s, t, INF, vis);
        if (pushed == 0) break;
        flow += pushed;
    }
    return flow;
}
```

Complexity: $(OE \cdot \text{max flow})$, depends on flow magnitude.

4. Edmonds-Karp Algorithm

A refinement:

> Always choose shortest augmenting path (by edges) using BFS.

Guarantees polynomial time.

A. Implementation (BFS + parent tracking)

```
int bfs(int s, int t, vector<int>& parent, int n) {
    fill(parent.begin(), parent.end(), -1);
    queue<pair<int,int>> q;
    q.push({s, INF});
    parent[s] = -2;
    while (!q.empty()) {
        auto [u, flow] = q.front(); q.pop();
        for (auto [v, _] : adj[u]) {
            if (parent[v] == -1 && cap[u][v] > 0) {
                int new_flow = min(flow, cap[u][v]);
                parent[v] = u;
                if (v == t) return new_flow;
                q.push({v, new_flow});
            }
        }
    }
    return 0;
}

int edmonds_karp(int s, int t, int n) {
    int flow = 0;
    vector<int> parent(n + 1);
    int new_flow;
    while ((new_flow = bfs(s, t, parent, n))) {
        flow += new_flow;
        int v = t;
        while (v != s) {
            int u = parent[v];
            cap[u][v] -= new_flow;
            cap[v][u] += new_flow;
            v = u;
        }
    }
    return flow;
}
```

Complexity: (OVE^2) Always terminates (no dependence on flow values).

5. Dinic's Algorithm

A modern classic , uses BFS to build level graph, and DFS to send blocking flow.

It works layer-by-layer, avoiding useless exploration.

A. Steps

1. Build level graph via BFS (assign levels to reachable nodes).
2. DFS sends flow along level-respecting paths.
3. Repeat until no path remains.

B. Implementation

```cpp
vector<int> level, ptr;

bool bfs_level(int s, int t, int n) {
    fill(level.begin(), level.end(), -1);
    queue<int> q;
    q.push(s);
    level[s] = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (auto [v, _] : adj[u])
            if (level[v] == -1 && cap[u][v] > 0) {
                level[v] = level[u] + 1;
                q.push(v);
            }
    }
    return level[t] != -1;
}

int dfs_flow(int u, int t, int pushed) {
    if (u == t || pushed == 0) return pushed;
    for (int &cid = ptr[u]; cid < (int)adj[u].size(); cid++) {
        int v = adj[u][cid].first;
        if (level[v] == level[u] + 1 && cap[u][v] > 0) {
            int tr = dfs_flow(v, t, min(pushed, cap[u][v]));
            if (tr > 0) {
                cap[u][v] -= tr;
                cap[v][u] += tr;
                return tr;
            }
        }
    }
    return 0;
}

int dinic(int s, int t, int n) {
    int flow = 0;
    level.resize(n + 1);
    ptr.resize(n + 1);
    while (bfs_level(s, t, n)) {
        fill(ptr.begin(), ptr.end(), 0);
        while (int pushed = dfs_flow(s, t, INF))
            flow += pushed;
    }
    return flow;
}
```

Complexity: (OEV^2) worst case, $(OE\sqrt{V})$ in practice.

6. Comparison

Algorithm	Strategy	Handles	Time	Notes
Ford-Fulkerson	DFS augmenting paths	Integral capacities	$OE \times max_flow$	Simple, may loop on reals
Edmonds-Karp	BFS augmenting paths	All capacities	$O(VE^2)$	Always terminates
Dinic	Level graph + DFS	All capacities	$O(V^2E)$	Fast in practice

7. Applications

- Network routing- Bipartite matching- Task assignment (flows = people → jobs)- Image segmentation (min-cut)- Circulation with demands- Data pipelines, max throughput systems

Tiny Code

Ford-Fulkerson Example:

```
add_edge(1, 2, 3);
add_edge(1, 3, 2);
add_edge(2, 3, 5);
add_edge(2, 4, 2);
add_edge(3, 4, 3);
printf("Max flow = %d\n", ford_fulkerson(1, 4, 4)); // 5
```

Why It Matters

Flow algorithms transform capacity constraints into solvable systems. They reveal the deep unity between optimization and structure: every maximum flow defines a minimum bottleneck cut.

They embody a timeless truth:

> "To understand limits, follow the flow."

Try It Yourself

1. Implement Ford-Fulkerson using DFS.
2. Switch to Edmonds-Karp and observe performance gain.
3. Build Dinic's level graph and visualize layers.
4. Model job assignment as bipartite flow.
5. Verify Max Flow = Min Cut on small examples.

Once you master flows, you'll see them hidden in everything that moves , from data to decisions.

37. Cuts (Stoer-Wagner, Karger, Gomory-Hu)

Where flow problems ask *"How much can we send?"*, cut problems ask *"Where does it break?"*

A cut splits a graph into two disjoint sets. The minimum cut is the smallest set of edges whose removal disconnects the graph , the tightest "bottleneck" holding it together.

This chapter explores three major algorithms:

- Stoer-Wagner , deterministic min-cut for undirected graphs- Karger's Randomized Algorithm , fast, probabilistic- Gomory-Hu Tree , compress all-pairs min-cuts into one tree Cuts reveal hidden structure , clusters, vulnerabilities, boundaries , and form the dual to flows via the Max-Flow Min-Cut Theorem.

1. The Min-Cut Problem

Given a weighted undirected graph (G = (V, E)): Find the minimum total weight of edges whose removal disconnects the graph.

Equivalent to:

> The smallest sum of edge weights crossing any partition ($S, V \setminus S$).

For directed graphs, you use max-flow methods; For undirected graphs, specialized algorithms exist.

2. Applications

- Network reliability , weakest link detection- Clustering , partition graph by minimal interconnection- Circuit design , splitting components- Image segmentation , separating regions- Community detection , sparse connections between groups

3. Stoer-Wagner Algorithm (Deterministic)

A clean, deterministic method for global minimum cut in undirected graphs.

A. Idea

1. Start with the full vertex set (V).
2. Repeatedly run Maximum Adjacency Search:

   - Start from a vertex - Grow a set by adding the most tightly connected vertex - The last added vertex defines a cut3. Contract the last two added vertices into one.

3. Keep track of smallest cut seen.

Repeat until one vertex remains.

B. Implementation (Adjacency Matrix)

```
const int INF = 1e9;
int g[MAX][MAX], w[MAX];
bool added[MAX], exist[MAX];

int stoer_wagner(int n) {
    int best = INF;
    vector<int> v(n);
    iota(v.begin(), v.end(), 0);

    while (n > 1) {
        fill(w, w + n, 0);
        fill(added, added + n, false);
        int prev = 0;
        for (int i = 0; i < n; i++) {
            int sel = -1;
            for (int j = 0; j < n; j++)
                if (!added[j] && (sel == -1 || w[j] > w[sel])) sel = j;
            if (i == n - 1) {
                best = min(best, w[sel]);
                for (int j = 0; j < n; j++)
                    g[prev][j] = g[j][prev] += g[sel][j];
                v.erase(v.begin() + sel);
                n--;
                break;
            }
            added[sel] = true;
            for (int j = 0; j < n; j++) w[j] += g[sel][j];
            prev = sel;
        }
    }
    return best;
}
```

Complexity: (OV^3), or $(OVE + V^2 \log V)$ with heaps Input: weighted undirected graph Output: global min cut value

C. Example

Graph:

```
1 -3- 2
|     |
4     2
 \-5-/
```

Cuts:

- {1,2}|{3} → 7- {1,3}|{2} → 5 Min cut = 5

4. Karger's Algorithm (Randomized)

A simple, elegant probabilistic method. Repeatedly contract random edges until two vertices remain; the remaining crossing edges form a cut.

Run multiple times $\rightarrow$ high probability of finding min cut.

A. Algorithm

1. While (|V| > 2):

   - Choose random edge ((u, v)) - Contract (u, v) into one node - Remove self-loops2. Return number of edges between remaining nodes

Repeat $(\mathrm{O}n^2 \log n)$ times for high confidence.

B. Implementation Sketch

```
struct Edge { int u, v; };
vector<Edge> edges;
int parent[MAX];

int find(int x) { return parent[x] == x ? x : parent[x] = find(parent[x]); }
void unite(int a, int b) { parent[find(b)] = find(a); }

int karger(int n) {
    int m = edges.size();
    iota(parent, parent + n, 0);
    int vertices = n;
    while (vertices > 2) {
        int i = rand() % m;
        int u = find(edges[i].u), v = find(edges[i].v);
        if (u == v) continue;
        unite(u, v);
        vertices--;
    }
    int cuts = 0;
    for (auto e : edges)
        if (find(e.u) != find(e.v)) cuts++;
    return cuts;
}
```

Expected Time: (On^2) per run Probability of success: (2 / (n(n-1))) per run Run multiple trials and take minimum.

C. Use Case

Great for large sparse graphs, or when approximate solutions are acceptable. Intuitive: the min cut survives random contractions if chosen carefully enough.

5. Gomory-Hu Tree

A compact way to store all-pairs min-cuts. It compresses (OV^2) flow computations into V-1 cuts.

A. Idea

- Build a tree where the min cut between any two vertices = the minimum edge weight on their path in the tree.

B. Algorithm

1. Pick vertex (s).
2. For each vertex $t \neq s$,

    - Run max flow to find min cut between (s, t). - Partition vertices accordingly.3. Connect partitions to form a tree.

Result: Gomory-Hu tree (V-1 edges).

Now any pair's min cut = smallest edge on path between them.

Complexity: (O(V)) max flow runs.

C. Uses

- Quickly answer all-pairs cut queries- Network reliability- Hierarchical clustering

6. Comparison

Algorithm	Type	Randomized	Graph	Complexity	Output
Stoer-Wagner	Deterministic	No	Undirected	$O(V^3)$	Global min cut
Karger	Randomized	Yes	Undirected	$O(n^2 \log n)$ (multi-run)	Probabilistic min cut
Gomory-Hu	Deterministic	No	Undirected	$O(V \times \text{MaxFlow})$	All-pairs min cuts

7. Relationship to Flows

By Max-Flow Min-Cut, min-cut capacity = max-flow value.

So you can find:

- s-t min cut = via max flow- global min cut = min over all (s, t) pairs Specialized algorithms just make it faster.

Tiny Code

Stoer-Wagner Example:

```c
printf("Global Min Cut = %d\n", stoer_wagner(n));
```

Karger Multi-Run:

```c
int ans = INF;
for (int i = 0; i < 100; i++)
    ans = min(ans, karger(n));
printf("Approx Min Cut = %d\n", ans);
```

Why It Matters

Cuts show you fragility , the weak seams of connection. While flows tell you *how much can pass*, cuts reveal *where it breaks first.*

They teach:

> "To understand strength, study what happens when you pull things apart."

Try It Yourself

1. Implement Stoer-Wagner and test on small graphs.
2. Run Karger 100 times and track success rate.
3. Build a Gomory-Hu tree and answer random pair queries.
4. Verify Max-Flow = Min-Cut equivalence on examples.
5. Use cuts for community detection in social graphs.

Mastering cuts gives you both grip and insight , where systems hold, and where they give way.

38. Matchings (Hopcroft-Karp, Hungarian, Blossom)

In many problems, we need to pair up elements efficiently: students to schools, jobs to workers, tasks to machines.

These are matching problems , find sets of edges with no shared endpoints that maximize cardinality or weight.

Depending on graph type, different algorithms apply:

- Hopcroft-Karp , fast matching in bipartite graphs- Hungarian Algorithm , optimal weighted assignment- Edmonds' Blossom Algorithm , general graphs (non-bipartite) Matching is a fundamental combinatorial structure, appearing in scheduling, flow networks, and resource allocation.

1. Terminology

- Matching: set of edges with no shared vertices- Maximum Matching: matching with largest number of edges- Perfect Matching: covers all vertices (each vertex matched once)- Maximum Weight Matching: matching with largest total edge weight Graph Types:
- Bipartite: vertices split into two sets (L, R); edges only between sets- General: arbitrary connections (may contain odd cycles)

2. Applications

- Job assignment- Network flows- Resource allocation- Student-project pairing- Stable marriages (with preferences)- Computer vision (feature correspondence)

3. Hopcroft-Karp Algorithm (Bipartite Matching)

A highly efficient algorithm for maximum cardinality matching in bipartite graphs.

It uses layered BFS + DFS to find multiple augmenting paths simultaneously.

A. Idea

1. Initialize matching empty.
2. While augmenting paths exist:
    - BFS builds layer graph (shortest augmenting paths). - DFS finds all augmenting paths along those layers. Each phase increases matching size significantly.

B. Complexity

$$O(E\sqrt{V})$$

Much faster than augmenting one path at a time (like Ford-Fulkerson).

C. Implementation

Let `pairU[u]` = matched vertex in R, or 0 if unmatched `pairV[v]` = matched vertex in L, or 0 if unmatched

```
vector<int> adjL[MAX];
int pairU[MAX], pairV[MAX], dist[MAX];
int nL, nR;

bool bfs() {
    queue<int> q;
    for (int u = 1; u <= nL; u++) {
        if (!pairU[u]) dist[u] = 0, q.push(u);
        else dist[u] = INF;
    }
    int found = INF;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        if (dist[u] < found) {
            for (int v : adjL[u]) {
                if (!pairV[v]) found = dist[u] + 1;
                else if (dist[pairV[v]] == INF) {
                    dist[pairV[v]] = dist[u] + 1;
                    q.push(pairV[v]);
                }
            }
        }
    }
    return found != INF;
}

bool dfs(int u) {
    for (int v : adjL[u]) {
        if (!pairV[v] || (dist[pairV[v]] == dist[u] + 1 && dfs(pairV[v]))) {
            pairU[u] = v;
            pairV[v] = u;
            return true;
        }
    }
    dist[u] = INF;
    return false;
}

int hopcroft_karp() {
    int matching = 0;
    while (bfs()) {
        for (int u = 1; u <= nL; u++)
            if (!pairU[u] && dfs(u)) matching++;
    }
    return matching;
}
```

D. Example

Graph:

```
U = {1,2,3}, V = {a,b}
Edges: 1-a, 2-a, 3-b
```

Matching: {1-a, 3-b} (size 2)

4. Hungarian Algorithm (Weighted Bipartite Matching)

Solves assignment problem , given cost matrix c_{ij}, assign each (i) to one (j) minimizing total cost (or maximizing profit).

A. Idea

Subtract minimums row- and column-wise $\rightarrow$ expose zeros $\rightarrow$ find minimal zero-cover $\rightarrow$ adjust matrix $\rightarrow$ repeat.

Equivalent to solving min-cost perfect matching on a bipartite graph.

B. Complexity

$$O(V^3)$$

Works for dense graphs, moderate sizes.

C. Implementation Sketch (Matrix Form)

```
int hungarian(const vector<vector<int>>& cost) {
    int n = cost.size();
    vector<int> u(n+1), v(n+1), p(n+1), way(n+1);
    for (int i = 1; i <= n; i++) {
        p[0] = i; int j0 = 0;
        vector<int> minv(n+1, INF);
        vector<char> used(n+1, false);
        do {
            used[j0] = true;
            int i0 = p[j0], delta = INF, j1;
            for (int j = 1; j <= n; j++) if (!used[j]) {
                int cur = cost[i0-1][j-1] - u[i0] - v[j];
                if (cur < minv[j]) minv[j] = cur, way[j] = j0;
                if (minv[j] < delta) delta = minv[j], j1 = j;
            }
            for (int j = 0; j <= n; j++)
                if (used[j]) u[p[j]] += delta, v[j] -= delta;
                else minv[j] -= delta;
            j0 = j1;
        } while (p[j0]);
        do { int j1 = way[j0]; p[j0] = p[j1]; j0 = j1; } while (j0);
    }
    return -v[0]; // minimal cost
}
```

D. Example

Cost matrix:

```
  a  b  c
1 3  2  1
2 2  3  2
3 3  2  3
```

Optimal assignment = 1-c, 2-a, 3-b Cost = 1 + 2 + 2 = 5

5. Edmonds' Blossom Algorithm (General Graphs)

For non-bipartite graphs, simple augmenting path logic breaks down (odd cycles). Blossom algorithm handles this via contraction of blossoms (odd cycles).

A. Idea

- Find augmenting paths- When odd cycle encountered (blossom), shrink it into one vertex- Continue search- Expand blossoms at end

B. Complexity

$$O(V^3)$$

Though complex to implement, it's the general-purpose solution for matchings.

C. Use Cases

- Non-bipartite job/task assignments- General pairing problems- Network design

6. Comparison

Algorithm	Graph Type	Weighted	Complexity	Output
Hopcroft-Karp	Bipartite	No	$O(E\sqrt{V})$	Max cardinality
Hungarian	Bipartite	Yes	$O(V^3)$	Min/Max cost matching
Blossom	General	Yes	$O(V^3)$	Max cardinality or weight

7. Relation to Flows

Bipartite matching = max flow on network:

- Left → Source edges (capacity 1)- Right → Sink edges (capacity 1)- Between sets → edges (capacity 1) Matching size = flow value

Tiny Code

Hopcroft-Karp Demo:

```
nL = 3; nR = 2;
adjL[1] = {1};
adjL[2] = {1};
adjL[3] = {2};
printf("Max Matching = %d\n", hopcroft_karp()); // 2
```

Why It Matters

Matchings are the language of pairing and assignment. They express cooperation without overlap , a structure of balance.

They reveal a deep duality:

> "Every match is a flow, every assignment an optimization."

Try It Yourself

1. Build a bipartite graph and run Hopcroft-Karp.
2. Solve an assignment problem with Hungarian algorithm.
3. Explore Blossom's contraction idea conceptually.
4. Compare max-flow vs matching approach.
5. Use matching to model scheduling (people   tasks).

Matching teaches how to pair without conflict, a lesson both mathematical and universal.

39. Tree Algorithms (LCA, HLD, Centroid Decomposition)

Trees are the backbone of many algorithms , they are connected, acyclic, and wonderfully structured.

Because of their simplicity, they allow elegant divide-and-conquer, dynamic programming, and query techniques. This section covers three fundamental patterns:

- Lowest Common Ancestor (LCA) , answer ancestor queries fast- Heavy-Light Decomposition (HLD) , break trees into chains for segment trees / path queries- Centroid Decomposition , recursively split tree by balance for divide-and-conquer Each reveals a different way to reason about trees , by depth, by chains, or by balance.

1. Lowest Common Ancestor (LCA)

Given a tree, two nodes (u, v). The LCA is the lowest node (farthest from root) that is an ancestor of both.

Applications:

- Distance queries- Path decomposition- RMQ / binary lifting- Tree DP and rerooting

A. Naive Approach

Climb ancestors until they meet. But this is (O(n)) per query , too slow for many queries.

B. Binary Lifting

Precompute ancestors at powers of 2. Then jump up by powers to align depths.

Preprocessing:

1. DFS to record depth
2. `up[v][k]` = 2^k-th ancestor of v

Answering query:

1. Lift deeper node up to same depth
2. Lift both together while `up[u][k] != up[v][k]`
3. Return parent

Code:

```
const int LOG = 20;
vector<int> adj[MAX];
int up[MAX][LOG], depth[MAX];

void dfs(int u, int p) {
    up[u][0] = p;
    for (int k = 1; k < LOG; k++)
        up[u][k] = up[up[u][k-1]][k-1];
    for (int v : adj[u]) if (v != p) {
        depth[v] = depth[u] + 1;
        dfs(v, u);
    }
}

int lca(int u, int v) {
    if (depth[u] < depth[v]) swap(u, v);
    int diff = depth[u] - depth[v];
    for (int k = 0; k < LOG; k++)
        if (diff & (1 << k)) u = up[u][k];
    if (u == v) return u;
    for (int k = LOG-1; k >= 0; k--)
        if (up[u][k] != up[v][k])
            u = up[u][k], v = up[v][k];
    return up[u][0];
}
```

Complexity:

- Preprocess: $(On \log n)$- Query: $(O\log n)$

C. Example

Tree:

```
    1
   / \
  2   3
 / \
4   5
```

- LCA(4,5) = 2- LCA(4,3) = 1

2. Heavy-Light Decomposition (HLD)

When you need to query paths (sum, max, min, etc.) on trees efficiently, you can use Heavy-Light Decomposition.

A. Idea

Decompose the tree into chains:

- Heavy edge = edge to child with largest subtree- Light edges = others Result: Every path from root to leaf crosses at most $(O\log n)$ light edges.

So, a path query can be broken into $(O\log^2 n)$ segment tree queries.

B. Steps

1. DFS to compute subtree sizes and identify heavy child
2. Decompose into chains
3. Assign IDs for segment tree
4. Use Segment Tree / BIT on linearized array

Key functions:

- `dfs_sz(u)` $\rightarrow$ compute subtree sizes- `decompose(u, head)` $\rightarrow$ assign chain heads Code (core):

```
int parent[MAX], depth[MAX], heavy[MAX], head[MAX], pos[MAX];
int cur_pos = 0;

int dfs_sz(int u) {
    int size = 1, max_sz = 0;
    for (int v : adj[u]) if (v != parent[u]) {
        parent[v] = u;
        depth[v] = depth[u] + 1;
        int sz = dfs_sz(v);
        if (sz > max_sz) max_sz = sz, heavy[u] = v;
        size += sz;
    }
    return size;
}

void decompose(int u, int h) {
    head[u] = h;
    pos[u] = cur_pos++;
    if (heavy[u] != -1) decompose(heavy[u], h);
    for (int v : adj[u])
        if (v != parent[u] && v != heavy[u])
            decompose(v, v);
}
```

Query path(u, v):

- While heads differ, move up chain by chain- Query segment tree in `[pos[head[u]], pos[u]]`- When in same chain, query segment `[pos[v], pos[u]]` Complexity:
- Build: (O(n))- Query/Update: ($O\log^2 n$)

C. Use Cases

- Path sums- Path maximums- Edge updates- Subtree queries

3. Centroid Decomposition

Centroid = node that splits tree into subtrees n/2 each. By removing centroids recursively, we form a centroid tree.

Used for divide-and-conquer on trees.

A. Steps

1. Find centroid

   - DFS to compute subtree sizes - Choose node where largest subtree n/22. Decompose:
   - Remove centroid - Recurse on subtrees Code (core):

```cpp
int subtree[MAX];
bool removed[MAX];
vector<int> adj[MAX];

int dfs_size(int u, int p) {
    subtree[u] = 1;
    for (int v : adj[u])
        if (v != p && !removed[v])
            subtree[u] += dfs_size(v, u);
    return subtree[u];
}

int find_centroid(int u, int p, int n) {
    for (int v : adj[u])
        if (v != p && !removed[v])
            if (subtree[v] > n / 2)
                return find_centroid(v, u, n);
    return u;
}

void decompose(int u, int p) {
    int n = dfs_size(u, -1);
    int c = find_centroid(u, -1, n);
    removed[c] = true;
    // process centroid here
    for (int v : adj[c])
        if (!removed[v])
            decompose(v, c);
}
```

Complexity: $(On \log n)$

B. Applications

- Distance queries (decompose + store distance to centroid)- Tree problems solvable by divide-and-conquer- Dynamic queries (add/remove nodes)

4. Comparison

Algorithm	Purpose	Query	Preprocess	Complexity	Notes
LCA	Ancestor query	$(O\log n)$	$(On\log n)$	Fast ancestor lookup	
HLD	Path queries	$(O\log^2 n)$	(O(n))	Segment tree-friendly	
Centroid Decomposition	Divide tree	-	$(On\log n)$	Balanced splits	

5. Interconnections

- HLD often uses LCA internally.- Centroid decomposition may use distance to ancestor (via LCA).- All exploit tree structure to achieve sublinear queries.

Tiny Code

LCA(4,5):

```
dfs(1,1);
printf("%d\n", lca(4,5)); // 2
```

HLD Path Sum: Build segment tree on `pos[u]` order, query along chains.

Centroid: `decompose(1, -1);`

Why It Matters

Tree algorithms show how structure unlocks efficiency. They transform naive traversals into fast, layered, or recursive solutions.

To master data structures, you must learn to "climb" and "cut" trees intelligently.

> "Every rooted path hides a logarithm."

Try It Yourself

1. Implement binary lifting LCA and test queries.
2. Add segment tree over HLD and run path sums.
3. Decompose tree by centroid and count nodes at distance k.
4. Combine LCA + HLD for path min/max.
5. Draw centroid tree of a simple graph.

Master these, and trees will stop being "just graphs" , they'll become *tools*.

40. Advanced Graph Algorithms and Tricks

By now you've seen the big families , traversals, shortest paths, flows, matchings, cuts, and trees. But real-world graphs often bring extra constraints: dynamic updates, multiple sources, layered structures, or special properties (planar, DAG, sparse).

This section gathers powerful advanced graph techniques , tricks and patterns that appear across problems once you've mastered the basics.

We'll explore:

- Topological Sorting & DAG DP- Strongly Connected Components (Condensation Graphs)- Articulation Points & Bridges (2-Edge/Vertex Connectivity)- Eulerian & Hamiltonian Paths- Graph Coloring & Bipartiteness Tests- Cycle Detection & Directed Acyclic Reasoning- Small-to-Large Merging, DSU on Tree, Mo's Algorithm on Trees- Bitmask DP on Graphs- Dynamic Graphs (Incremental/Decremental BFS/DFS)- Special Graphs (Planar, Sparse, Dense) These aren't just algorithms , they're patterns that let you attack harder graph problems with insight.

1. Topological Sorting & DAG DP

In a DAG (Directed Acyclic Graph), edges always point forward. This makes it possible to order vertices linearly so all edges go from left to right , a topological order.

Use cases:

- Task scheduling- Dependency resolution- DP on DAG (longest/shortest path, counting paths) Algorithm (Kahn's):

```
vector<int> topo_sort(int n) {
    vector<int> indeg(n+1), res;
    queue<int> q;
    for (int u = 1; u <= n; u++)
        for (int v : adj[u]) indeg[v]++;
    for (int u = 1; u <= n; u++)
        if (!indeg[u]) q.push(u);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        res.push_back(u);
        for (int v : adj[u])
            if (--indeg[v] == 0) q.push(v);
    }
    return res;
}
```

DAG DP:

```
vector<int> dp(n+1, 0);
for (int u : topo_order)
    for (int v : adj[u])
        dp[v] = max(dp[v], dp[u] + weight(u,v));
```

Complexity: O(V + E)

2. Strongly Connected Components (Condensation)

In directed graphs, vertices may form SCCs (mutually reachable components). Condensing SCCs yields a DAG, often easier to reason about.

Use:

- Component compression- Meta-graph reasoning- Cycle condensation Tarjan's Algorithm: DFS with low-link values, single pass.

Kosaraju's Algorithm: Two passes , DFS on graph and reversed graph.

Complexity: O(V + E)

Once SCCs are built, you can run DP or topological sort on the condensed DAG.

3. Articulation Points & Bridges

Find critical vertices/edges whose removal disconnects the graph.

- Articulation point: vertex whose removal increases component count- Bridge: edge whose removal increases component count Algorithm: Tarjan's DFS Track discovery time `tin[u]` and lowest reachable ancestor `low[u]`.

```
void dfs(int u, int p) {
    tin[u] = low[u] = ++timer;
    for (int v : adj[u]) {
        if (v == p) continue;
        if (!tin[v]) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] > tin[u]) bridge(u, v);
            if (low[v] >= tin[u] && p != -1) cut_vertex(u);
        } else low[u] = min(low[u], tin[v]);
    }
}
```

Applications:

- Network reliability- Biconnected components- 2-edge/vertex connectivity tests

4. Eulerian & Hamiltonian Paths

- Eulerian Path: visits every edge exactly once
    - Exists if graph is connected and 0 or 2 vertices have odd degree- Hamiltonian Path: visits every vertex exactly once (NP-hard) Euler Tour Construction: Hierholzer's algorithm (O(E))

Applications:

- Route reconstruction (e.g., word chains)- Postman problems

5. Graph Coloring & Bipartiteness

Bipartite Check: DFS/ BFS alternating colors Fails if odd cycle found.

```
bool bipartite(int n) {
    vector<int> color(n+1, -1);
    for (int i = 1; i <= n; i++) if (color[i] == -1) {
        queue<int> q; q.push(i); color[i] = 0;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (int v : adj[u]) {
                if (color[v] == -1)
                    color[v] = color[u] ^ 1, q.push(v);
                else if (color[v] == color[u])
                    return false;
            }
        }
    }
    return true;
}
```

Applications:

- 2-SAT reduction- Planar graph coloring- Conflict-free assignment

6. Cycle Detection

- DFS + recursion stack for directed graphs- Union-Find for undirected graphs Used to test acyclicity, detect back edges, or find cycles for rollback or consistency checks.

7. DSU on Tree (Small-to-Large Merging)

For queries like “count distinct colors in subtree,” merge results from smaller to larger subtrees to maintain O(n log n).

Pattern:

1. DFS through children
2. Keep large child’s data structure
3. Merge small child’s data in

Applications:

- Offline subtree queries- Heavy subproblem caching

8. Mo's Algorithm on Trees

Offline algorithm to answer path queries efficiently:

- Convert path queries to ranges via Euler Tour- Use Mo's ordering to process in O((N + Q)$\sqrt{}$N) Useful when online updates aren't required.

9. Bitmask DP on Graphs

For small graphs (n   20): State = subset of vertices e.g., Traveling Salesman Problem (TSP)

```
dp[mask][u] = min cost to visit mask, end at u
```

Transition:

```
dp[mask | (1<<v)][v] = min(dp[mask][u] + cost[u][v])
```

Complexity: O(n^2 2)

10. Dynamic Graphs

Graphs that change:

- Incremental BFS: maintain distances as edges added- Decremental connectivity: union-find rollback or dynamic trees Used in online queries, evolving networks, or real-time systems.

11. Special Graph Classes

- Planar graphs: 3V-6E; use face counting- Sparse graphs: adjacency lists best- Dense graphs: adjacency matrix / bitset Optimizations often hinge on density.

Tiny Code

Topological Order:

```
auto order = topo_sort(n);
for (int u : order) printf("%d ", u);
```

Bridge Check: `if (low[v] > tin[u])` edge is a bridge.

Euler Path Check: Count odd-degree nodes == 0 or 2.

Why It Matters

These advanced techniques complete your toolkit. They're not isolated , they combine to solve real-world puzzles: dependency graphs, robust networks, optimized paths, compressed states.

They teach a mindset:

> "Graphs are not obstacles , they're shapes of possibility."

Try It Yourself

1. Implement topological sort and DAG DP.
2. Find SCCs and build condensation graph.
3. Detect articulation points and bridges.
4. Check Euler path conditions on random graphs.
5. Try DSU on tree for subtree statistics.
6. Solve TSP via bitmask DP for n   15.

Once you can mix and match these tools, you're no longer just navigating graphs , you're shaping them.

Chapter 5. Dynamic Programming

41. DP Basics and State Transitions

Dynamic Programming (DP) is one of the most powerful ideas in algorithm design. It's about breaking a big problem into smaller overlapping subproblems, solving each once, and reusing their answers.

When brute force explodes exponentially, DP brings it back under control. This section introduces the mindset, the mechanics, and the math behind DP.

1. The Core Idea

Many problems have two key properties:

- Overlapping subproblems: The same smaller computations repeat many times.
- Optimal substructure: The optimal solution to a problem can be built from optimal solutions to its subproblems.

DP solves each subproblem once, stores the result, and reuses it. This saves exponential time , often reducing (O2^n) to (On^2) or (O(n)).

2. The Recipe

When approaching a DP problem, follow this pattern:

1. Define the state. Decide what subproblems you'll solve. Example: `dp[i] = best answer for first i elements`.
2. Write the recurrence. Express each state in terms of smaller ones. Example: `dp[i] = dp[i-1] + cost(i)`
3. Set the base cases. Where does the recursion start? Example: `dp[0] = 0`
4. Decide the order. Bottom-up (iterative) or top-down (recursive with memoization).
5. Return the final answer. Often `dp[n]` or `max(dp[i])`.

3. Example: Fibonacci Numbers

Let's begin with a classic , the nth Fibonacci number (F(n) = F(n-1) + F(n-2)).

Recursive (slow):

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

This recomputes the same values over and over , exponential time.

Top-Down DP (Memoization):

```
int dp[MAX];
int fib(int n) {
    if (n <= 1) return n;
    if (dp[n] != -1) return dp[n];
    return dp[n] = fib(n-1) + fib(n-2);
}
```

Bottom-Up DP (Tabulation):

```
int fib(int n) {
    int dp[n+1];
    dp[0] = 0; dp[1] = 1;
    for (int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}
```

Space Optimized:

```
int fib(int n) {
    int a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

4. States, Transitions, and Dependencies

A DP table is a map from states to answers. Each state depends on others via a transition function.

Think of it like a graph , each edge represents a recurrence relation.

Example:

- State: `dp[i] = number of ways to reach step i`- Transition: `dp[i] = dp[i-1] + dp[i-2]` (like stairs)- Base: `dp[0] = 1`

5. Common DP Patterns

1. 1D Linear DP
   - Problems like Fibonacci, climbing stairs, LIS.
2. 2D DP
   - Grids, sequences, or combinations (LCS, knapsack).
3. Bitmask DP

    - Subsets, TSP, combinatorial optimization.

4. DP on Trees
    - Subtree computations (sum, diameter).
5. Digit DP
    - Counting numbers with properties in a range.
6. Segment DP
    - Matrix chain multiplication, interval merges.

6. Top-Down vs Bottom-Up

Approach	Method	Pros	Cons
Top-Down	Recursion + Memoization	Easy to write, intuitive	Stack overhead, needs memo
Bottom-Up	Iteration	Fast, space-optimizable	Harder to derive order

When dependencies are simple and acyclic, bottom-up shines. When they're complex, top-down is easier.

7. Example 2: Climbing Stairs

You can climb 1 or 2 steps at a time. How many distinct ways to reach step (n)?

State: `dp[i] = ways to reach step i` Transition: `dp[i] = dp[i-1] + dp[i-2]` Base: `dp[0] = 1, dp[1] = 1`

Code:

```
int climb(int n) {
    int dp[n+1];
    dp[0] = dp[1] = 1;
    for (int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}
```

8. Debugging DP

To debug DP:

- Print intermediate states.- Visualize table (especially 2D).- Check base cases.- Trace one small example by hand.

9. Complexity

Most DP algorithms are linear or quadratic in number of states:

- Time = (#states) × (work per transition)- Space = (#states) Example: Fibonacci: (O(n)) time, (O(1)) space Knapsack: ($On \times W$) LCS: ($On \times m$)

Tiny Code

Fibonacci (tabulated):

```
int dp[100];
dp[0] = 0; dp[1] = 1;
for (int i = 2; i <= n; i++)
    dp[i] = dp[i-1] + dp[i-2];
printf("%d", dp[n]);
```

Why It Matters

DP is the art of remembering. It transforms recursion into iteration, chaos into order.

From optimization to counting, from paths to sequences , once you see substructure, DP becomes your hammer.

> "Every repetition hides a recurrence."

Try It Yourself

1. Write top-down and bottom-up Fibonacci.
2. Count ways to climb stairs with steps {1,2,3}.
3. Compute number of paths in an n×m grid.
4. Try to spot state, recurrence, base in each problem.
5. Draw dependency graphs to visualize transitions.

DP isn't a formula , it's a mindset: break problems into parts, remember the past, and build from it.

42. Classic Problems (Knapsack, Subset Sum, Coin Change)

Now that you know what dynamic programming *is*, let's dive into the classic trio , problems that every programmer meets early on:

- Knapsack (maximize value under weight constraint)- Subset Sum (can we form a given sum?)- Coin Change (how many ways or fewest coins to reach a total) These are the training grounds of DP: each shows how to define states, transitions, and base cases clearly.

1. 0/1 Knapsack Problem

Problem: You have `n` items, each with weight `w[i]` and value `v[i]`. A knapsack with capacity `W`. Pick items (each at most once) to maximize total value, without exceeding weight.

A. State

`dp[i][w]` = max value using first `i` items with capacity `w`

B. Recurrence

For item `i`:

- If we don't take it: `dp[i-1][w]`- If we take it (if `w[i]`   `w`): `dp[i-1][w - w[i]] + v[i]` So,
$$dp[i][w] = \max(dp[i-1][w], dp[i-1][w-w[i]] + v[i])$$

C. Base Case

`dp[0][w] = 0` for all w (no items = no value)

D. Implementation

```
int knapsack(int n, int W, int w[], int v[]) {
    int dp[n+1][W+1];
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= W; j++) {
            if (i == 0 || j == 0) dp[i][j] = 0;
            else if (w[i-1] <= j)
                dp[i][j] = max(dp[i-1][j], dp[i-1][j - w[i-1]] + v[i-1]);
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    return dp[n][W];
}
```

Complexity: Time: (O(nW)) Space: (O(nW)) (can be optimized to 1D (O(W)))

E. Space Optimization (1D DP)

```
int dp[W+1] = {0};
for (int i = 0; i < n; i++)
    for (int w = W; w >= weight[i]; w--)
        dp[w] = max(dp[w], dp[w - weight[i]] + value[i]);
```

F. Example

Items:

```
w = [2, 3, 4, 5]
v = [3, 4, 5, 6]
W = 5
```

Best: take items 1 + 2 → value 7

2. Subset Sum

Problem: Given a set `S` of integers, can we pick some to sum to `target`?

A. State

`dp[i][sum] = true` if we can form sum `sum` using first `i` elements.

B. Recurrence

- Don't take: `dp[i-1][sum]`- Take (if `a[i]` `sum`): `dp[i-1][sum - a[i]]` So,

$$dp[i][sum] = dp[i-1][sum]; ||; dp[i-1][sum - a[i]]$$

C. Base Case

`dp[0][0] = true` (sum 0 possible with no elements) `dp[0][sum] = false` for sum > 0

D. Implementation

```
bool subset_sum(int a[], int n, int target) {
    bool dp[n+1][target+1];
    for (int i = 0; i <= n; i++) dp[i][0] = true;
    for (int j = 1; j <= target; j++) dp[0][j] = false;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= target; j++) {
            if (a[i-1] > j) dp[i][j] = dp[i-1][j];
            else dp[i][j] = dp[i-1][j] || dp[i-1][j - a[i-1]];
        }
    }
    return dp[n][target];
}
```

Complexity: Time: $(On \cdot target)$

E. Example

S = [3, 34, 4, 12, 5, 2], target = 9 Yes $\rightarrow$ 4 + 5

3. Coin Change

Two variants:

(a) Count Ways (Unbounded Coins)

"How many ways to make total `T` with coins `c[]`?"

Order doesn't matter.

State: `dp[i][t] = ways using first i coins for total t`

Recurrence:

- Skip coin: `dp[i-1][t]`- Take coin (unlimited): `dp[i][t - c[i]]`

$$dp[i][t] = dp[i-1][t] + dp[i][t-c[i]]$$

Base: `dp[0][0] = 1`

1D Simplified:

```
int dp[T+1] = {0};
dp[0] = 1;
for (int coin : coins)
    for (int t = coin; t <= T; t++)
        dp[t] += dp[t - coin];
```

(b) Min Coins (Fewest Coins to Reach Total)

State: `dp[t] = min coins to reach t`

Recurrence:

$$dp[t] = \min_{c_i \le t}(dp[t-c_i] + 1)$$

Base: `dp[0] = 0`, rest = INF

```
int dp[T+1];
fill(dp, dp+T+1, INF);
dp[0] = 0;
for (int t = 1; t <= T; t++)
    for (int c : coins)
        if (t >= c) dp[t] = min(dp[t], dp[t - c] + 1);
```

Example

Coins = [1,2,5], Total = 5

- Ways: 4 (5; 2+2+1; 2+1+1+1; 1+1+1+1+1)- Min Coins: 1 (5)

4. Summary

Problem	Type	State	Transition	Complexity
0/1 Knapsack	Max value	dp[i][w]	max(take, skip)	O(nW)
Subset Sum	Feasibility	dp[i][sum]	OR of include/exclude	O(n * sum)
Coin Change (ways)	Counting	dp[t]	dp[t] + dp[t - coin]	O(nT)
Coin Change (min)	Optimization	dp[t]	min(dp[t - coin] + 1)	O(nT)

Tiny Code

Min Coin Change (1D):

```
int dp[T+1];
fill(dp, dp+T+1, INF);
dp[0] = 0;
for (int c : coins)
    for (int t = c; t <= T; t++)
        dp[t] = min(dp[t], dp[t - c] + 1);
printf("%d\n", dp[T]);
```

Why It Matters

These three are archetypes:

- Knapsack: optimize under constraint- Subset Sum: choose feasibility- Coin Change: count or minimize Once you master them, you can spot their patterns in harder problems , from resource allocation to pathfinding.

  "Every constraint hides a choice; every choice hides a state."

Try It Yourself

1. Implement 0/1 Knapsack (2D and 1D).
2. Solve Subset Sum for target 30 with random list.
3. Count coin combinations for amount 10.
4. Compare "min coins" vs "ways to form."
5. Write down state-transition diagram for each.

These three form your DP foundation , the grammar for building more complex algorithms.

43. Sequence Problems (LIS, LCS, Edit Distance)

Sequence problems form the *heart* of dynamic programming. They appear in strings, arrays, genomes, text comparison, and version control. Their power comes from comparing prefixes , building large answers from aligned smaller ones.

This section explores three cornerstones:

- LIS (Longest Increasing Subsequence)- LCS (Longest Common Subsequence)- Edit Distance (Levenshtein Distance) Each teaches a new way to think about subproblems, transitions, and structure.

1. Longest Increasing Subsequence (LIS)

Problem: Given an array, find the length of the longest subsequence that is *strictly increasing.*

A subsequence isn't necessarily contiguous , you can skip elements.

Example: `[10, 9, 2, 5, 3, 7, 101, 18]` → LIS is `[2, 3, 7, 18]` → length 4

A. State

`dp[i]` = length of LIS ending at index `i`

B. Recurrence

$$dp[i] = 1 + \max_{j<i \wedge a[j]<a[i]} dp[j]$$

If no smaller `a[j]`, then `dp[i] = 1`.

C. Base

`dp[i] = 1` for all i (each element alone is an LIS)

D. Implementation

```
int lis(int a[], int n) {
    int dp[n], best = 0;
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        for (int j = 0; j < i; j++)
            if (a[j] < a[i])
                dp[i] = max(dp[i], dp[j] + 1);
        best = max(best, dp[i]);
    }
    return best;
}
```

Complexity: (On^2)

E. Binary Search Optimization

Use a tail array:

- `tail[len] = min possible ending value of LIS of length len` For each `x`:
- Replace `tail[idx]` via lower_bound

```
int lis_fast(vector<int>& a) {
    vector<int> tail;
    for (int x : a) {
        auto it = lower_bound(tail.begin(), tail.end(), x);
        if (it == tail.end()) tail.push_back(x);
        else *it = x;
    }
    return tail.size();
}
```

Complexity: $(On \log n)$

2. Longest Common Subsequence (LCS)

Problem: Given two strings, find the longest subsequence present in both.

Example: `s1 = "ABCBDAB"`, `s2 = "BDCABA"` LCS = “BCBA” → length 4

A. State

`dp[i][j]` = LCS length between `s1[0..i-1]` and `s2[0..j-1]`

B. Recurrence

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1, & \text{if } s_1[i-1] = s_2[j-1], \\ \max(dp[i-1][j],\, dp[i][j-1]), & \text{otherwise.} \end{cases}$$

C. Base

`dp[0][*] = dp[*][0] = 0` (empty string)

D. Implementation

```
int lcs(string a, string b) {
    int n = a.size(), m = b.size();
    int dp[n+1][m+1];
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++)
            if (i == 0 || j == 0) dp[i][j] = 0;
            else if (a[i-1] == b[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[n][m];
}
```

Complexity: (O(nm))

E. Reconstruct LCS

Trace back from `dp[n][m]`:

- If chars equal → take it and move diagonally- Else move toward larger neighbor

F. Example

a = "AGGTAB", b = "GXTXAYB" LCS = "GTAB" → 4

3. Edit Distance (Levenshtein Distance)

Problem: Minimum operations (insert, delete, replace) to convert string `a` $\to$ `b`.

Example: `kitten → sitting` $= 3$ (replace k$\to$s, insert i, insert g)

A. State

`dp[i][j]` $=$ min edits to convert `a[0..i-1]` $\to$ `b[0..j-1]`

B. Recurrence

If `a[i-1] == b[j-1]`:

$$dp[i][j] = dp[i-1][j-1]$$

Else:

$$dp[i][j] = 1 + \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])$$

(Delete, Insert, Replace)

C. Base

- `dp[0][j] = j` (insert all)- `dp[i][0] = i` (delete all)

D. Implementation

```
int edit_distance(string a, string b) {
    int n = a.size(), m = b.size();
    int dp[n+1][m+1];
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++) {
            if (i == 0) dp[i][j] = j;
            else if (j == 0) dp[i][j] = i;
            else if (a[i-1] == b[j-1])
                dp[i][j] = dp[i-1][j-1];
            else
                dp[i][j] = 1 + min({dp[i-1][j], dp[i][j-1], dp[i-1][j-1]});
        }
    return dp[n][m];
}
```

Complexity: (O(nm))

E. Example

a = "horse", b = "ros"

- replace h→r, delete r, delete e → 3

4. Summary

Problem	Type	State	Transition	Complexity
LIS	Single seq	dp[i]	1 + max(dp[j])	$O(n^2)$ / O(n log n)
LCS	Two seqs	dp[i][j]	if match +1 else max	O(nm)
Edit Distance	Two seqs	dp[i][j]	if match 0 else 1 + min	O(nm)

5. Common Insights

- LIS builds upward , from smaller sequences.- LCS aligns two sequences , compare prefixes.- Edit Distance quantifies *difference* , minimal edits. They're templates for bioinformatics, text diffing, version control, and more.

Tiny Code

LCS:

```
if (a[i-1] == b[j-1])
    dp[i][j] = dp[i-1][j-1] + 1;
else
    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
```

Why It Matters

Sequence DPs teach you how to compare progressions , how structure and similarity evolve over time.

They transform vague "compare these" tasks into crisp recurrence relations.

> "To align is to understand."

Try It Yourself

1. Implement LIS (O(n^2) and O(n log n))
2. Find LCS of two given strings
3. Compute edit distance between "intention" and "execution"
4. Modify LCS to print one valid subsequence
5. Try to unify LCS and Edit Distance in a single table

Master these, and you can handle any DP on sequences , the DNA of algorithmic thinking.

44. Matrix and Chain Problems

Dynamic programming shines when a problem involves choices over intervals , which order, which split, which parenthesis. This chapter explores a class of problems built on chains and matrices, where order matters and substructure is defined by intervals.

We'll study:

- Matrix Chain Multiplication (MCM) - optimal parenthesization- Polygon Triangulation - divide shape into minimal-cost triangles- Optimal BST / Merge Patterns - weighted merging decisions These problems teach interval DP, where each state represents a segment ([i, j]).

1. Matrix Chain Multiplication (MCM)

Problem: Given matrices $A_1, A_2, ..., A_n$, find the parenthesization that minimizes total scalar multiplications.

Matrix A_i has dimensions $p[i-1] \times p[i]$. We can multiply $A_i \cdot A_{i+1}$ only if inner dimensions match.

Goal: Minimize operations:

$$\text{cost}(i,j) = \min_k \left(\text{cost}(i,k) + \text{cost}(k+1,j) + p[i-1] \cdot p[k] \cdot p[j]\right)$$

A. State

`dp[i][j]` = min multiplications to compute $A_i ... A_j$

B. Base

`dp[i][i] = 0` (single matrix needs no multiplication)

C. Recurrence

$$dp[i][j] = \min_{i \le k < j} dp[i][k] + dp[k+1][j] + p[i-1] \times p[k] \times p[j]$$

D. Implementation

```
int matrix_chain(int p[], int n) {
    int dp[n][n];
    for (int i = 1; i < n; i++) dp[i][i] = 0;

    for (int len = 2; len < n; len++) {
        for (int i = 1; i + len - 1 < n; i++) {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;
            for (int k = i; k < j; k++)
                dp[i][j] = min(dp[i][j],
                    dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j]);
        }
    }
    return dp[1][n-1];
}
```

Complexity: $(\mathrm{O}n^3)$ time, $(\mathrm{O}n^2)$ space

E. Example

p = [10, 20, 30, 40, 30] Optimal order: ((A1A2)A3)A4 → cost 30000

2. Polygon Triangulation

Given a convex polygon with `n` vertices, connect non-intersecting diagonals to minimize total cost. Cost of a triangle = perimeter or product of side weights.

This is the same structure as MCM , divide polygon by diagonals.

A. State

`dp[i][j]` = min triangulation cost for polygon vertices from i to j.

B. Recurrence

$$dp[i][j] = \min_{i<k<j}(dp[i][k] + dp[k][j] + cost(i, j, k))$$

Base: dp[i][i+1] = 0 (fewer than 3 points)

C. Implementation

```
double polygon_triangulation(vector<Point> &p) {
    int n = p.size();
    double dp[n][n];
    for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) dp[i][j] = 0;
    for (int len = 2; len < n; len++) {
        for (int i = 0; i + len < n; i++) {
            int j = i + len;
            dp[i][j] = 1e18;
            for (int k = i+1; k < j; k++)
                dp[i][j] = min(dp[i][j],
                    dp[i][k] + dp[k][j] + dist(p[i],p[k])+dist(p[k],p[j])+dist(p[j],p[i]));
        }
    }
    return dp[0][n-1];
}
```

Complexity: (On^3)

3. Optimal Binary Search Tree (OBST)

Given sorted keys $k_1 < k_2 < \cdots < k_n$ with search frequencies (f[i]), construct a BST with minimal expected search cost.

The more frequently accessed nodes should be nearer the root.

A. State

`dp[i][j]` = min cost to build BST from keys `i..j` `sum[i][j]` = sum of frequencies from i to j (precomputed)

B. Recurrence

$$dp[i][j] = \min_{k=i}^{j}(dp[i][k-1] + dp[k+1][j] + sum[i][j])$$

Each root adds one to depth of its subtrees → extra cost = `sum[i][j]`

C. Implementation

```
int optimal_bst(int freq[], int n) {
    int dp[n][n], sum[n][n];
    for (int i = 0; i < n; i++) {
        dp[i][i] = freq[i];
        sum[i][i] = freq[i];
        for (int j = i+1; j < n; j++)
            sum[i][j] = sum[i][j-1] + freq[j];
    }
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i+len-1 < n; i++) {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;
            for (int r = i; r <= j; r++) {
                int left = (r > i) ? dp[i][r-1] : 0;
                int right = (r < j) ? dp[r+1][j] : 0;
                dp[i][j] = min(dp[i][j], left + right + sum[i][j]);
            }
        }
    }
    return dp[0][n-1];
}
```

Complexity: (On^3)

4. Merge Pattern Problems

Many problems , merging files, joining ropes, Huffman coding , involve repeatedly combining elements with minimal total cost.

All follow this template:

$$dp[i][j] = \min_{k}(dp[i][k] + dp[k+1][j] + \text{merge cost})$$

Same structure as MCM.

5. Key Pattern: Interval DP

State: `dp[i][j] = best answer for subarray [i..j]` Transition: Try all splits `k` between `i` and `j`

Template:

```
for (len = 2; len <= n; len++)
 for (i = 0; i + len - 1 < n; i++) {
    j = i + len - 1;
    dp[i][j] = INF;
    for (k = i; k < j; k++)
       dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + cost(i,j,k));
 }
```

6. Summary

Problem	State	Recurrence	Complexity
MCM	dp[i][j]	min(dp[i][k]+dp[k+1][j]+p[i-1]*p[k]*p[j])	O(n^3)
Polygon Triangulation	dp[i][j]	min(dp[i][k]+dp[k][j]+cost)	O(n^3)
OBST	dp[i][j]	min(dp[i][k-1]+dp[k+1][j]+sum[i][j])	O(n^3)
Merge Problems	dp[i][j]	min(dp[i][k]+dp[k+1][j]+merge cost)	O(n^3)

Tiny Code

Matrix Chain (Compact):

```
for (len = 2; len < n; len++)
  for (i = 1; i + len - 1 < n; i++) {
    j = i + len - 1; dp[i][j] = INF;
    for (k = i; k < j; k++)
      dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j]);
  }
```

Why It Matters

These problems are DP in 2D , reasoning over intervals and splits. They train your ability to "cut the problem" at every possible point.

> "Between every start and end lies a choice of where to divide."

Try It Yourself

1. Implement MCM and print parenthesization.
2. Solve polygon triangulation with edge weights.
3. Build OBST for frequencies [34, 8, 50].
4. Visualize DP table diagonally.
5. Generalize to merging `k` segments at a time.

Master these, and you'll see interval DP patterns hiding in parsing, merging, and even AI planning.

45. Bitmask DP and Traveling Salesman

Some dynamic programming problems require you to track which items have been used, or which subset of elements is active at a given point. This is where Bitmask DP shines. It encodes subsets as binary masks, allowing you to represent state space efficiently.

This technique is a must-know for:

- Traveling Salesman Problem (TSP)- Subset covering / visiting problems- Permutations and combinations of sets- Game states and toggles

1. The Idea of Bitmask DP

A bitmask is an integer whose binary representation encodes a subset.

For (n) elements:

- There are 2^n subsets.- A subset is represented by a mask from `0` to `(1 << n) - 1`. Example for `n = 4`:

Subset	Mask (binary)	Mask (decimal)
	0000	0
{0}	0001	1
{1}	0010	2
{0,1,3}	1011	11

We can check membership:

- `mask & (1 << i)` → whether element `i` is in subset We can add elements:
- `mask | (1 << i)` → add element `i` We can remove elements:
- `mask & ~(1 << i)` → remove element `i`

2. Example: Traveling Salesman Problem (TSP)

Problem: Given `n` cities and cost matrix `cost[i][j]`, find the minimum cost Hamiltonian cycle visiting all cities exactly once and returning to start.

A. State

`dp[mask][i]` = minimum cost to reach city `i` having visited subset `mask`

- `mask` → set of visited cities- `i` → current city

B. Base Case

`dp[1<<0][0] = 0` (start at city 0, only 0 visited)

C. Transition

For each subset `mask` and city `i` in `mask`, try moving from `i` to `j` not in `mask`:

$$dp[mask \cup (1 << j)][j] = \min\left(dp[mask \cup (1 << j)][j], dp[mask][i] + cost[i][j]\right)$$

D. Implementation

```
int tsp(int n, int cost[20][20]) {
    int N = 1 << n;
    const int INF = 1e9;
    int dp[N][n];
    for (int m = 0; m < N; m++)
        for (int i = 0; i < n; i++)
            dp[m][i] = INF;

    dp[1][0] = 0; // start at city 0

    for (int mask = 1; mask < N; mask++) {
        for (int i = 0; i < n; i++) {
            if (!(mask & (1 << i))) continue;
            for (int j = 0; j < n; j++) {
                if (mask & (1 << j)) continue;
                int next = mask | (1 << j);
                dp[next][j] = min(dp[next][j], dp[mask][i] + cost[i][j]);
            }
        }
    }

    int ans = INF;
    for (int i = 1; i < n; i++)
        ans = min(ans, dp[N-1][i] + cost[i][0]);
    return ans;
}
```

Complexity:

- States: ($On \cdot 2^n$)- Transitions: (O(n))- Total: ($On^2 \cdot 2^n$)

E. Example

```
n = 4
cost = {
 {0, 10, 15, 20},
 {10, 0, 35, 25},
 {15, 35, 0, 30},
 {20, 25, 30, 0}
}
```

Optimal path: $0 \to 1 \to 3 \to 2 \to 0$ Cost $= 80$

3. Other Common Bitmask DP Patterns

1. Subset Sum / Partition `dp[mask] = true if subset represented by mask satisfies property`
2. Counting Set Bits `__builtin_popcount(mask)` gives number of elements in subset.
3. Iterating Over Submasks

```
for (int sub = mask; sub; sub = (sub-1) & mask)
    // handle subset sub
```

4. Assigning Tasks (Assignment Problem)

- Each mask represents set of workers assigned.- State: `dp[mask] = min cost for assigned tasks`.

```
for (mask) for (task)
 if (!(mask & (1 << task)))
   dp[mask | (1 << task)] = min(dp[mask | (1 << task)],
        dp[mask] + cost[__builtin_popcount(mask)][task]);
```

4. Memory Tricks

- If only previous masks needed, use rolling arrays:

```
dp[next][j] = ...
swap(dp, next_dp)
```

- Compress dimensions: $(O2^n)$ memory for small `n`

5. Summary

Problem	State	Transition	Complexity
TSP	dp[mask][i]	min(dp[mask][i] + cost[i][j])	$O(n^2 \cdot 2\ )$
Assignment	dp[mask]	add one new element	$O(n^2 \cdot 2\ )$
Subset Sum	dp[mask]	union of valid subsets	$O(2\ \cdot n)$

Tiny Code

Core Transition:

```
for (mask)
  for (i)
    if (mask & (1<<i))
      for (j)
        if (!(mask & (1<<j)))
          dp[mask|(1<<j)][j] = min(dp[mask|(1<<j)][j], dp[mask][i] + cost[i][j]);
```

Why It Matters

Bitmask DP is how you enumerate subsets efficiently. It bridges combinatorics and optimization, solving exponential problems with manageable constants.

> "Every subset is a story, and bits are its alphabet."

Try It Yourself

1. Solve TSP with 4 cities (hand-trace the table).
2. Implement Assignment Problem using bitmask DP.
3. Count subsets with even sum.
4. Use bitmask DP to find maximum compatible set of tasks.
5. Explore how to optimize memory with bit tricks.

Bitmask DP unlocks the world of subset-based reasoning , the foundation of combinatorial optimization.

46. Digit DP and SOS DP

In some problems, you don't iterate over indices or subsets , you iterate over digits or masks to count or optimize over structured states. Two major flavors stand out:

- Digit DP - counting numbers with certain properties (e.g. digit sum, constraints)- SOS DP (Sum Over Subsets) - efficiently computing functions over all subsets These are essential techniques when brute force would require enumerating every number or subset, which quickly becomes impossible.

1. Digit DP (Counting with Constraints)

Digit DP is used to count or sum over all numbers N that satisfy a condition, such as:

- The sum of digits equals a target.- The number doesn't contain a forbidden digit.- The number has certain parity or divisibility. Instead of iterating over all numbers (up to 10^1 !), we iterate digit-by-digit.

A. State Design

Typical DP state:

`dp[pos][sum][tight][leading_zero]`

- `pos`: current digit index (from most significant to least)- `sum`: property tracker (e.g. sum of digits, remainder)- `tight`: whether we're still restricted by N's prefix- `leading_zero`: whether we've started placing nonzero digits

B. Transition

At each digit position, we choose a digit `d`:

```
limit = tight ? (digit at pos in N) : 9
for (d = 0; d <= limit; d++) {
    new_tight = tight && (d == limit)
    new_sum = sum + d
    // or new_mod = (mod * 10 + d) % M
}
```

Transition accumulates results across valid choices.

C. Base Case

When `pos == len(N)` (end of digits):

- Return 1 if condition holds (e.g. `sum == target`), else 0

D. Example: Count numbers N with digit sum = S

```
long long dp[20][200][2];

long long solve(string s, int pos, int sum, bool tight) {
    if (pos == s.size()) return sum == 0;
    if (sum < 0) return 0;
    if (dp[pos][sum][tight] != -1) return dp[pos][sum][tight];

    int limit = tight ? (s[pos] - '0') : 9;
    long long res = 0;
    for (int d = 0; d <= limit; d++)
        res += solve(s, pos+1, sum-d, tight && (d==limit));

    return dp[pos][sum][tight] = res;
}
```

Usage:

```
string N = "12345";
int S = 9;
memset(dp, -1, sizeof dp);
cout << solve(N, 0, S, 1);
```

Complexity: O(number of digits $\times$ sum $\times$ 2) $\to$ typically O(20 $\times$ 200 $\times$ 2)

E. Example Variants

1. Count numbers divisible by 3 → track remainder: `new_rem = (rem*10 + d) % 3`
2. Count numbers without consecutive equal digits → add `last_digit` to state.
3. Count beautiful numbers (like palindromes, no repeated digits) → track bitmask of used digits.

F. Summary

Problem	State	Transition	Complexity
Sum of digits = S	dp[pos][sum][tight]	sum-d	O(len · S)
Divisible by k	dp[pos][rem][tight]	(rem*10+d)%k	O(len · k)
No repeated digits	dp[pos][mask][tight]	mask	O(len · 2^{1})

Tiny Code

```
for (int d = 0; d <= limit; d++)
    res += solve(pos+1, sum-d, tight && (d==limit));
```

2. SOS DP (Sum Over Subsets)

When dealing with functions on subsets, we sometimes need to compute:

$$f(S) = \sum_{T \subseteq S} g(T)$$

Naively O(3). SOS DP reduces it to O(n · 2).

A. Setup

Let `f[mask] = g[mask]` initially. For each bit `i`:

```
for (mask = 0; mask < (1<<n); mask++)
    if (mask & (1<<i))
        f[mask] += f[mask^(1<<i)];
```

After this, `f[mask]` = sum of `g[sub]` for all `sub   mask`.

B. Example

Given array `a[mask]`, compute `sum[mask] = sum_{sub   mask} a[sub]`

```
int n = 3;
int N = 1 << n;
int f[N], a[N];
// initialize a[]
for (int mask = 0; mask < N; mask++) f[mask] = a[mask];
for (int i = 0; i < n; i++)
  for (int mask = 0; mask < N; mask++)
    if (mask & (1 << i))
        f[mask] += f[mask ^ (1 << i)];
```

C. Why It Works

Each iteration adds contributions from subsets differing by one bit. By processing all bits, every subset's contribution propagates upward.

D. Variants

- Sum over supersets: reverse direction.- Max instead of sum: replace `+=` with `max=`.- XOR convolution: combine values under XOR subset relation.

E. Applications

- Inclusion-exclusion acceleration- Precomputing subset statistics- DP over masks with subset transitions

F. Complexity

Problem	Naive	SOS DP
Subset sum	O(3)	O(n · 2)
Superset sum	O(3)	O(n · 2)

Why It Matters

Digit DP teaches counting under constraints , thinking digit by digit. SOS DP teaches subset propagation , spreading information efficiently.

Together, they show how to tame exponential state spaces with structure.

> "When the search space explodes, symmetry and structure are your compass."

Try It Yourself

1. Count numbers 10 whose digit sum = 10.
2. Count numbers 10 without repeated digits.
3. Compute f[mask] = sum_{sub mask} a[sub] for n=4.
4. Use SOS DP to find how many subsets of bits have even sum.
5. Modify Digit DP to handle leading zeros explicitly.

Master these, and you can handle structured exponential problems with elegance and speed.

47. DP Optimizations (Divide & Conquer, Convex Hull Trick, Knuth)

Dynamic Programming often starts with a simple recurrence, but naïve implementations can be too slow (e.g., (On^2) or worse). When the recurrence has special structure , such as monotonicity or convexity , we can exploit it to reduce time complexity drastically.

This chapter introduces three powerful optimization families:

1. Divide and Conquer DP
2. Convex Hull Trick (CHT)
3. Knuth Optimization

Each one is based on discovering order or geometry hidden inside transitions.

1. Divide and Conquer Optimization

If you have a recurrence like:

$$dp[i] = \min_{k<i} dp[k] + C(k, i)$$

and the optimal k for dp[i]   optimal k for dp[i+1], you can use divide & conquer to compute dp in (O$n \log n$) or (O$n \log^2 n$).

This property is called monotonicity of argmin.

A. Conditions

Let (C(k, i)) be the cost to transition from (k) to (i). Divide and conquer optimization applies if:

$$opt(i) \leq opt(i+1)$$

and (C) satisfies quadrangle inequality (or similar convex structure).

B. Template

```
void compute(int l, int r, int optL, int optR) {
    if (l > r) return;
    int mid = (l + r) / 2;
    pair<long long,int> best = {INF, -1};
    for (int k = optL; k <= min(mid, optR); k++) {
        long long val = dp_prev[k] + cost(k, mid);
        if (val < best.first) best = {val, k};
    }
    dp[mid] = best.first;
    int opt = best.second;
    compute(l, mid-1, optL, opt);
    compute(mid+1, r, opt, optR);
}
```

You call it as:

```
compute(1, n, 0, n-1);
```

C. Example: Divide Array into K Segments

Given array `a[1..n]`, divide into `k` parts to minimize

$$dp[i][k] = \min_{j<i} dp[j][k-1] + cost(j+1, i)$$

If cost satisfies quadrangle inequality, you can optimize each layer in (O$n \log n$).

D. Complexity

Naive: (On^2) $\to$ Optimized: (O$n \log n$)

2. Convex Hull Trick (CHT)

Applies when DP recurrence is linear in i and k:

$$dp[i] = \min_{k<i}(m_k \cdot x_i + b_k)$$

where:

- m_k is slope (depends on k)- (b_k = dp[k] + c(k))- x_i is known (monotonic) You can maintain lines $y = m_k x + b_k$ in a convex hull, and query min efficiently.

A. Conditions

- Slopes m_k are monotonic (either increasing or decreasing)- Query points x_i are sorted If both monotonic, we can use pointer walk in O(1) amortized per query. Otherwise, use Li Chao Tree (O(log n)).

B. Implementation (Monotonic Slopes)

```
struct Line { long long m, b; };
deque<Line> hull;

bool bad(Line l1, Line l2, Line l3) {
    return (l3.b - l1.b)*(l1.m - l2.m) <= (l2.b - l1.b)*(l1.m - l3.m);
}

void add(long long m, long long b) {
    Line l = {m, b};
    while (hull.size() >= 2 && bad(hull[hull.size()-2], hull.back(), l))
        hull.pop_back();
    hull.push_back(l);
}

long long query(long long x) {
    while (hull.size() >= 2 &&
          hull[0].m*x + hull[0].b >= hull[1].m*x + hull[1].b)
        hull.pop_front();
    return hull.front().m*x + hull.front().b;
}
```

C. Example: DP for Line-Based Recurrence

$$dp[i] = a_i^2 + \min_{j<i}(dp[j] + b_j \cdot a_i)$$

Here $m_j = b_j$, $x_i = a_i$, $b_j = dp[j]$

D. Complexity

- Naive: (On^2)- CHT: (O(n)) or ($On \log n$)

3. Knuth Optimization

Used in interval DP problems (like Matrix Chain, Merging Stones).

If:

1. $dp[i][j] = \min_{k=i}^{j-1}(dp[i][k] + dp[k+1][j] + w(i,j))$
2. The cost $w(i,j)$ satisfies the quadrangle inequality:

$$w(a,c) + w(b,d) \leq w(a,d) + w(b,c)$$

3. And the monotonicity condition:

$$opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$$

Then you can reduce the search space from (O(n)) to (O(1)) per cell, making total complexity ($\mathrm{O}n^2$) instead of ($\mathrm{O}n^3$).

A. Implementation

```
for (int len = 2; len <= n; len++) {
    for (int i = 1; i + len - 1 <= n; i++) {
        int j = i + len - 1;
        dp[i][j] = INF;
        for (int k = opt[i][j-1]; k <= opt[i+1][j]; k++) {
            long long val = dp[i][k] + dp[k+1][j] + cost(i,j);
            if (val < dp[i][j]) {
                dp[i][j] = val;
                opt[i][j] = k;
            }
        }
    }
}
```

B. Example

Optimal Binary Search Tree or Merging Stones (with additive cost). Typical improvement: ($\mathrm{O}n^3 \rightarrow \mathrm{O}n^2$)

4. Summary

Technique	Applies To	Key Property	Complexity
Divide & Conquer DP	1D transitions	Monotonic argmin	O(n log n)
Convex Hull Trick	Linear transitions	Monotonic slopes	O(n) / O(n log n)
Knuth Optimization	Interval DP	Quadrangle + Monotonicity	O(n^2)

Tiny Code

Divide & Conquer Template

```
void compute(int l, int r, int optL, int optR);
```

CHT Query

```
while (size>=2 && f[1](x) < f[0](x)) pop_front();
```

Why It Matters

These optimizations show that DP isn't just brute force with memory , it's mathematical reasoning on structure.

Once you spot monotonicity or linearity, you can shrink a quadratic solution into near-linear time.

> "Optimization is the art of seeing simplicity hiding in structure."

Try It Yourself

1. Optimize Matrix Chain DP using Knuth.
2. Apply Divide & Conquer on `dp[i] = min_{k<i}(dp[k]+(i-k)^2)`.
3. Solve Slope DP with CHT for convex cost functions.
4. Compare runtime vs naive DP on random data.
5. Derive conditions for opt monotonicity in your custom recurrence.

Master these techniques, and you'll turn your DPs from slow prototypes into lightning-fast solutions.

48. Tree DP and Rerooting

Dynamic Programming on trees is one of the most elegant and powerful patterns in algorithm design. Unlike linear arrays or grids, trees form hierarchical structures, where each node depends on its children or parent. Tree DP teaches you how to aggregate results up and down the tree, handling problems where subtrees interact.

In this section, we'll cover:

1. Basic Tree DP (rooted trees)
2. DP over children (bottom-up aggregation)
3. Rerooting technique (top-down propagation)
4. Common applications and examples

1. Basic Tree DP: The Idea

We define `dp[u]` to represent some property of the subtree rooted at `u`. Then we combine children's results to compute `dp[u]`.

This bottom-up approach is like postorder traversal.

Example structure:

```
function dfs(u, parent):
    dp[u] = base_value
    for v in adj[u]:
        if v == parent: continue
        dfs(v, u)
        dp[u] = combine(dp[u], dp[v])
```

Example 1: Size of Subtree

Let `dp[u] = number of nodes in subtree rooted at u`

```
void dfs(int u, int p) {
    dp[u] = 1;
    for (int v : adj[u]) {
        if (v == p) continue;
        dfs(v, u);
        dp[u] += dp[v];
    }
}
```

Key idea: Combine children's sizes to get parent size. Complexity: (O(n))

Example 2: Height of Tree

Let `dp[u] = height of subtree rooted at u`

```
void dfs(int u, int p) {
    dp[u] = 0;
    for (int v : adj[u]) {
        if (v == p) continue;
        dfs(v, u);
        dp[u] = max(dp[u], 1 + dp[v]);
    }
}
```

This gives you the height when rooted at any node.

2. DP Over Children (Bottom-Up Aggregation)

Tree DP is all about merging children.

For example, if you want the number of ways to color or number of independent sets, you compute children's dp and merge results at parent.

Example 3: Counting Independent Sets

In a tree, an independent set is a set of nodes with no two adjacent.

State:

- `dp[u][0]` = ways if `u` is not selected- `dp[u][1]` = ways if `u` is selected Recurrence:

$$dp[u][0] = \prod_{v \in children(u)} (dp[v][0] + dp[v][1])$$

$$dp[u][1] = \prod_{v \in children(u)} dp[v][0]$$

Implementation:

```
void dfs(int u, int p) {
    dp[u][0] = dp[u][1] = 1;
    for (int v : adj[u]) {
        if (v == p) continue;
        dfs(v, u);
        dp[u][0] *= (dp[v][0] + dp[v][1]);
        dp[u][1] *= dp[v][0];
    }
}
```

Final answer = `dp[root][0] + dp[root][1]`

Example 4: Maximum Path Sum in Tree

Let `dp[u]` = max path sum starting at `u` and going down To find best path anywhere, store a global max over child pairs.

```
int ans = 0;
int dfs(int u, int p) {
    int best1 = 0, best2 = 0;
    for (int v : adj[u]) {
        if (v == p) continue;
        int val = dfs(v, u) + weight(u, v);
        if (val > best1) swap(best1, val);
        if (val > best2) best2 = val;
    }
    ans = max(ans, best1 + best2);
    return best1;
}
```

This gives tree diameter or max path sum.

3. Rerooting Technique

Rerooting DP allows you to compute answers for every node as root, without recomputing from scratch (On^2). It's also known as DP on trees with re-rooting.

Idea

1. First, compute `dp_down[u]` = answer for subtree when rooted at `u`.
2. Then, propagate info from parent to child (`dp_up[u]`), so each node gets info from outside its subtree.
3. Combine `dp_down` and `dp_up` to get `dp_all[u]`.

Example 5: Sum of Distances from Each Node

Let's find `ans[u]` = sum of distances from `u` to all nodes.

1. Root the tree at 0.
2. Compute subtree sizes and total distance from root.
3. Reroot to adjust distances using parent's info.

Step 1: Bottom-up:

```
void dfs1(int u, int p) {
    sz[u] = 1;
    for (int v : adj[u]) {
        if (v == p) continue;
        dfs1(v, u);
        sz[u] += sz[v];
        dp[u] += dp[v] + sz[v];
    }
}
```

Step 2: Top-down:

```
void dfs2(int u, int p) {
    for (int v : adj[u]) {
        if (v == p) continue;
        dp[v] = dp[u] - sz[v] + (n - sz[v]);
        dfs2(v, u);
    }
}
```

After `dfs2`, `dp[u]` = sum of distances from node `u`.

Complexity: (O(n))

4. General Rerooting Template

```
// 1. Postorder: compute dp_down[u] from children
void dfs_down(u, p):
    dp_down[u] = base
    for v in adj[u]:
        if v != p:
            dfs_down(v, u)
            dp_down[u] = merge(dp_down[u], dp_down[v])

// 2. Preorder: use parent's dp_up to compute dp_all[u]
void dfs_up(u, p, dp_up_parent):
    ans[u] = merge(dp_down[u], dp_up_parent)
    prefix, suffix = prefix products of children
    for each child v:
        dp_up_v = merge(prefix[v-1], suffix[v+1], dp_up_parent)
        dfs_up(v, u, dp_up_v)
```

This template generalizes rerooting to many problems:

- Maximum distance from each node- Number of ways to select subtrees- Sum of subtree sizes seen from each root

5. Summary

Pattern	Description	Complexity
Basic Tree DP	Combine child subresults	O(n)
DP Over Children	Each node depends on children	O(n)
Rerooting DP	Compute result for every root	O(n)
Multiple States	Track choices (e.g. include/exclude)	O(n · state)

Tiny Code

Subtree Size

```
void dfs(int u, int p) {
    dp[u] = 1;
    for (int v: adj[u]) if (v != p) {
        dfs(v,u);
        dp[u] += dp[v];
    }
}
```

Reroot Sum Distances

```
dp[v] = dp[u] - sz[v] + (n - sz[v]);
```

Why It Matters

Tree DP is how we think recursively over structure , each node's truth emerges from its children. Rerooting expands this idea globally, giving every node its own perspective.

> "In the forest of states, each root sees a different world , yet all follow the same law."

Try It Yourself

1. Count number of nodes in each subtree.
2. Compute sum of depths from each node.
3. Find tree diameter using DP.
4. Count number of independent sets modulo 1e9+7.
5. Implement rerooting to find max distance from each node.

Tree DP turns recursive patterns into universal strategies for hierarchical data.

49. DP Reconstruction and Traceback

So far, we've focused on computing optimal values (min cost, max score, count of ways). But in most real problems, we don't just want the number , we want to know how we got it.

That's where reconstruction comes in: once you've filled your DP table, you can trace back the decisions that led to the optimal answer.

This chapter shows you how to:

1. Record transitions during DP computation
2. Reconstruct paths, subsets, or sequences
3. Handle multiple reconstructions (paths, sets, parent links)
4. Understand traceback in 1D, 2D, and graph-based DPs

1. The Core Idea

Each DP state comes from a choice. If you store *which choice was best*, you can walk backward from the final state to rebuild the solution.

Think of it as:

```
dp[i] = best over options
choice[i] = argmin or argmax option
```

Then:

```
reconstruction_path = []
i = n
while i > 0:
    reconstruction_path.push(choice[i])
    i = choice[i].prev
```

You're not just solving , you're remembering the path.

2. Reconstruction in 1D DP

Example: Coin Change (Minimum Coins)

Problem: Find minimum number of coins to make value `n`.

Recurrence:

$$dp[x] = 1 + \min_{c \in coins, c \leq x} dp[x-c]$$

To reconstruct which coins were used:

```
int dp[MAXN], prev_coin[MAXN];
dp[0] = 0;
for (int x = 1; x <= n; x++) {
    dp[x] = INF;
    for (int c : coins) {
        if (x >= c && dp[x-c] + 1 < dp[x]) {
            dp[x] = dp[x-c] + 1;
            prev_coin[x] = c;
        }
    }
}
```

Reconstruction:

```
vector<int> used;
int cur = n;
while (cur > 0) {
    used.push_back(prev_coin[cur]);
    cur -= prev_coin[cur];
}
```

Output: coins used in one optimal solution.

Example: LIS Reconstruction

You know how to find LIS length. Now reconstruct the sequence.

```
int dp[n], prev[n];
int best_end = 0;
for (int i = 0; i < n; i++) {
    dp[i] = 1; prev[i] = -1;
    for (int j = 0; j < i; j++)
        if (a[j] < a[i] && dp[j] + 1 > dp[i]) {
            dp[i] = dp[j] + 1;
            prev[i] = j;
        }
    if (dp[i] > dp[best_end]) best_end = i;
}
```

Rebuild LIS:

```
vector<int> lis;
for (int i = best_end; i != -1; i = prev[i])
    lis.push_back(a[i]);
reverse(lis.begin(), lis.end());
```

3. Reconstruction in 2D DP

Example: LCS (Longest Common Subsequence)

We have `dp[i][j]` filled using:

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1, & \text{if } a[i-1] = b[j-1], \\ \max(dp[i-1][j], dp[i][j-1]), & \text{otherwise.} \end{cases}$$

To reconstruct LCS:

```
int i = n, j = m;
string lcs = "";
while (i > 0 && j > 0) {
    if (a[i-1] == b[j-1]) {
        lcs.push_back(a[i-1]);
        i--; j--;
    }
    else if (dp[i-1][j] > dp[i][j-1]) i--;
    else j--;
}
reverse(lcs.begin(), lcs.end());
```

Output: one valid LCS string.

Example: Edit Distance

Operations: insert, delete, replace.

You can store the operation:

```
if (a[i-1] == b[j-1]) op[i][j] = "match";
else if (dp[i][j] == dp[i-1][j-1] + 1) op[i][j] = "replace";
else if (dp[i][j] == dp[i-1][j] + 1) op[i][j] = "delete";
else op[i][j] = "insert";
```

Then backtrack to list operations:

```
while (i > 0 || j > 0) {
    if (op[i][j] == "match") i--, j--;
    else if (op[i][j] == "replace") { print("Replace"); i--; j--; }
    else if (op[i][j] == "delete") { print("Delete"); i--; }
    else { print("Insert"); j--; }
}
```

4. Reconstruction in Path Problems

When DP tracks shortest paths, you can keep parent pointers.

Example: Bellman-Ford Path Reconstruction

```
int dist[n], parent[n];
dist[src] = 0;
for (int k = 0; k < n-1; k++)
  for (auto [u,v,w] : edges)
    if (dist[u] + w < dist[v]) {
        dist[v] = dist[u] + w;
        parent[v] = u;
    }

vector<int> path;
for (int v = dest; v != src; v = parent[v])
    path.push_back(v);
path.push_back(src);
reverse(path.begin(), path.end());
```

You now have the actual shortest path.

5. Handling Multiple Solutions

Sometimes multiple optimal paths exist. You can:

- Store all predecessors instead of one- Backtrack recursively to enumerate all solutions- Tie-break deterministically (e.g., lexicographically smallest) Example:

```
if (new_val == dp[i]) parents[i].push_back(j);
```

Then recursively generate all possible paths.

6. Visualization

DP reconstruction often looks like following arrows in a grid or graph:

- LCS: diagonal ( ), up (↑), left (←)- Shortest path: parent edges- LIS: predecessor chain You're walking through decisions, not just numbers.

7. Summary

Type	State	Reconstruction
1D DP	`prev[i]`	Trace chain
2D DP	`op[i][j]`	Follow choices
Graph DP	`parent[v]`	Follow edges
Counting DP	optional	Recover counts / paths

Tiny Code

General pattern:

```
for (state)
  for (choice)
    if (better) {
        dp[state] = value;
        parent[state] = choice;
    }
```

Then:

```
while (state != base) {
    path.push_back(parent[state]);
    state = parent[state];
}
```

Why It Matters

Solving DP gets you the score , reconstructing shows you the story. It's the difference between knowing the answer and understanding the reasoning.

> "Numbers tell you the outcome; pointers tell you the path."

Try It Yourself

1. Reconstruct one LIS path.
2. Print all LCSs for small strings.
3. Show edit operations to transform "cat" → "cut".
4. Track subset used in Knapsack to reach exact weight.

5. Recover optimal merge order in Matrix Chain DP.

Reconstruction turns DP from a static table into a narrative of decisions , a map back through the maze of optimal choices.

50. Meta-DP and Optimization Templates

We've now explored many flavors of dynamic programming , on sequences, grids, trees, graphs, subsets, and digits. This final chapter in the DP arc zooms out to the *meta-level*: how to see DP patterns, generalize them, and turn them into reusable templates.

If classical DP is about solving one problem, meta-DP is about recognizing *families* of problems that share structure. You'll learn how to build your own DP frameworks, use common templates, and reason from first principles.

1. What Is Meta-DP?

A *Meta-DP* is a high-level abstraction of a dynamic programming pattern. It encodes:

- State definition pattern- Transition pattern- Optimization structure- Dimensional dependencies By learning these patterns, you can design DPs faster, reuse logic across problems, and spot optimizations early.

Think of Meta-DP as:

> "Instead of memorizing 100 DPs, master 10 DP blueprints."

2. The Four Building Blocks

Every DP has the same core ingredients:

1. State: what subproblem you're solving

   - Often `dp[i]`, `dp[i][j]`, or `dp[mask]` - Represents smallest unit of progress2. Transition: how to build larger subproblems from smaller ones
   - E.g. `dp[i] = min(dp[j] + cost(j, i))`3. Base Case: known trivial answers
   - E.g. `dp[0] = 0`4. Order: how to fill the states
   - E.g. increasing `i`, decreasing `i`, or topological order Once you can describe a problem in these four, it *is* a DP.

3. Meta-Templates for Common Structures

Below are generalized templates to use and adapt.

A. Line DP (1D Sequential)

Shape: linear progression Examples:

- Fibonacci- Knapsack (1D capacity)- LIS (sequential dependency)

```
for (int i = 1; i <= n; i++) {
    dp[i] = base;
    for (int j : transitions(i))
        dp[i] = min(dp[i], dp[j] + cost(j, i));
}
```

Visualization: → → → Each state depends on previous positions.

B. Grid DP (2D Spatial)

Shape: grid or matrix Examples:

- Paths in a grid- Edit Distance- Counting paths with obstacles

```
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    dp[i][j] = combine(dp[i-1][j], dp[i][j-1]);
```

Visualization: Moves from top-left to bottom-right.

C. Interval DP

Shape: segments or subarrays Examples:

- Matrix Chain Multiplication- Optimal BST- Merging Stones

```
for (len = 2; len <= n; len++)
  for (i = 0; i + len - 1 < n; i++) {
      j = i + len - 1;
      dp[i][j] = INF;
      for (k = i; k < j; k++)
          dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + cost(i,j));
  }
```

Key Insight: overlapping intervals, split points.

D. Subset DP

Shape: subsets of a set Examples:

- Traveling Salesman (TSP)- Assignment problem- SOS DP

```
for (mask = 0; mask < (1<<n); mask++)
  for (sub = mask; sub; sub = (sub-1)&mask)
      dp[mask] = combine(dp[mask], dp[sub]);
```

Key Insight: use bitmasks to represent subsets.

E. Tree DP

Shape: hierarchical dependencies Examples:

- Subtree sizes- Independent sets- Rerooting

```
void dfs(u, p):
  dp[u] = base
  for (v in children)
    if (v != p)
      dfs(v, u)
      dp[u] = merge(dp[u], dp[v])
```

F. Graph DP (Topological Order)

Shape: DAG structure Examples:

- Longest path in DAG- Counting paths- DAG shortest path

```
for (u in topo_order)
  for (v in adj[u])
    dp[v] = combine(dp[v], dp[u] + weight(u,v));
```

Key: process nodes in topological order.

G. Digit DP

Shape: positional digits, constrained transitions Examples:

- Count numbers satisfying digit conditions- Divisibility / digit sum problems

```
dp[pos][sum][tight] =  dp[next_pos][new_sum][new_tight];
```

H. Knuth / Divide & Conquer / Convex Hull Trick

Shape: optimization over monotone or convex transitions Examples:

- Cost-based splits- Line-based transitions

```
dp[i] = min_k (dp[k] + cost(k, i))
```

Key: structure in `opt[i]` or `slope`.

Question	Clue

4. Recognizing DP Type

Ask these diagnostic questions:

Question	Clue
“Does each step depend on smaller subproblems?”	DP
“Do I split a segment?”	Interval DP
“Do I choose subsets?”	Subset / Bitmask DP
“Do I move along positions?”	Line DP
“Do I merge children?”	Tree DP
“Do I process in a DAG?”	Graph DP
“Do I track digits or constraints?”	Digit DP

5. Optimization Layer

Once you have a working DP, ask:

- Can transitions be reduced (monotonicity)?- Can overlapping cost be cached (prefix sums)?- Can dimensions be compressed (rolling arrays)?- Can you reuse solutions for each segment (Divide & Conquer / Knuth)? This transforms your DP from conceptual to efficient.

6. Meta-DP: Transformations

- Compress dimensions: if only `dp[i-1]` needed, use 1D array.- Invert loops: bottom-up top-down.- Change base: prefix-sums for range queries.- State lifting: add dimension for new property (like remainder, parity, bitmask). > “When stuck, add a dimension. When slow, remove one.”

7. Common Template Snippets

Rolling 1D Knapsack:

```
for (c = C; c >= w[i]; c--)
  dp[c] = max(dp[c], dp[c-w[i]] + val[i]);
```

Top-Down Memoization:

```
int solve(state):
  if (visited[state]) return dp[state];
  dp[state] = combine(solve(next_states));
```

Iterative DP:

```
for (state in order)
  dp[state] = merge(prev_states);
```

8. Building Your Own DP Framework

You can design a generic `DP(state, transition)` class:

```
struct DP {
    vector<long long> dp;
    function<long long(int,int)> cost;
    DP(int n, auto cost): dp(n, INF), cost(cost) {}
    void solve() { for (int i=1; i<n; i++) for (int j=0; j<i; j++)
        dp[i] = min(dp[i], dp[j] + cost(j, i)); }
};
```

Reusable, readable, flexible.

9. Summary

DP Type	Core State	Shape	Typical Complexity
Line DP	dp[i]	Linear	$O(n^2) \to O(n)$
Grid DP	dp[i][j]	Matrix	$O(n \cdot m)$
Interval DP	dp[i][j]	Triangular	$O(n^3)$
Subset DP	dp[mask]	Exponential	$O(n \cdot 2\ )$
Tree DP	dp[u]	Tree	$O(n)$
Digit DP	dp[pos][sum]	Recursive	$O(len \cdot sum)$
Graph DP	dp[v]	DAG	$O(V+E)$

Tiny Code

```
for (state in order)
  dp[state] = combine(all_prev(state));
```

Why It Matters

Meta-DP turns your thinking from case-by-case to pattern-by-pattern. You stop memorizing formulas and start *seeing shapes*: lines, grids, intervals, trees, masks.

Once you can name the shape, you can write the DP.

> "DP is not about filling tables. It's about recognizing structure."

Try It Yourself

1. Classify each classic DP problem into a type.
2. Write one template per pattern (Line, Grid, Tree, etc.).
3. Create a `dp_solve(state, transitions)` function to generalize logic.
4. For each pattern, practice a small example.
5. Build your own "Little Book of DP Patterns" with code snippets.

This is your bridge from concrete solutions to algorithmic fluency , the foundation for mastering the next 950 algorithms ahead.

Chapter 6. Strings and Text Algorithms

51. Number Theory (GCD, Modular Arithmetic, CRT)

Number theory forms the mathematical backbone of many algorithms , from hashing and cryptography to modular combinatorics and primality testing. In algorithmic problem-solving, it's all about working with integers, divisibility, and modular systems efficiently.

This section covers the essential toolkit:

- GCD and Extended Euclidean Algorithm- Modular Arithmetic (addition, subtraction, multiplication, inverse)- Modular Exponentiation- Chinese Remainder Theorem (CRT)

1. The Greatest Common Divisor (GCD)

The GCD of two integers a and b, denoted $\gcd(a, b)$, is the largest integer that divides both. It's the cornerstone for fraction simplification, Diophantine equations, and modular inverses.

A. Euclidean Algorithm

Based on:

$$\gcd(a,b) = \gcd(b, a \bmod b)$$

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

Time complexity: $O(\log \min(a,b))$

B. Extended Euclidean Algorithm

Finds integers (x, y) such that:

$$ax + by = \gcd(a,b)$$

This is critical for finding modular inverses.

```
int ext_gcd(int a, int b, int &x, int &y) {
    if (b == 0) {
        x = 1; y = 0;
        return a;
    }
    int x1, y1;
    int g = ext_gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return g;
}
```

C. Bezout's Identity

If $g = \gcd(a,b)$, then $ax + by = g$ has integer solutions. If $g = 1$, x is the modular inverse of $amodulob$.

2. Modular Arithmetic

A modular system “wraps around” after a certain value (m).

We write:

$$a \equiv b \pmod{m} \quad \text{if } m \mid (a-b)$$

It behaves like ordinary arithmetic, with the rule:

- $(a+b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$
- $(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$
- $(a-b) \bmod m = ((a \bmod m) - (b \bmod m) + m) \bmod m$

A. Modular Exponentiation

Compute $a^b \bmod m$ efficiently using binary exponentiation.

```
long long modpow(long long a, long long b, long long m) {
    long long res = 1;
    a %= m;
    while (b > 0) {
        if (b & 1) res = (res * a) % m;
        a = (a * a) % m;
        b >>= 1;
    }
    return res;
}
```

Complexity: ($\mathrm{O}\log b$)

B. Modular Inverse

Given (a), find a^{-1} such that:

$$a \cdot a^{-1} \equiv 1 \pmod{m}$$

Case 1: If (m) is prime, use Fermat’s Little Theorem:

$$a^{-1} \equiv a^{m-2} \pmod{m}$$

```
int modinv(int a, int m) {
    return modpow(a, m-2, m);
}
```

Case 2: If (a) and (m) are coprime, use Extended GCD:

```
int inv(int a, int m) {
    int x, y;
    int g = ext_gcd(a, m, x, y);
    if (g != 1) return -1; // no inverse
    return (x % m + m) % m;
}
```

C. Modular Division

To divide $a/b \bmod m$:

$$a/b \equiv a \cdot b^{-1} \pmod{m}$$

So compute the inverse and multiply.

3. Chinese Remainder Theorem (CRT)

The CRT solves systems of congruences:

$$x \equiv a_1 \pmod{m_1}$$

$$x \equiv a_2 \pmod{m_2}$$

If moduli $m_1, m_2, \dots, m_k$ are pairwise coprime, there exists a unique solution modulo $M = m_1 m_2 \dots m_k$.

A. 2-Equation Example

Solve:

$$x \equiv a_1 \pmod{m_1}, \quad x \equiv a_2 \pmod{m_2}$$

Let:

- $M = m_1 m_2$- $M_1 = M/m_1$- $M_2 = M/m_2$ Find inverses $inv_1 = M_1^{-1} \bmod m_1$, $inv_2 = M_2^{-1} \bmod m_2$

Then:

$$x = (a_1 \cdot M_1 \cdot inv_1 + a_2 \cdot M_2 \cdot inv_2) \bmod M$$

B. Implementation

```
long long crt(vector<int> a, vector<int> m) {
    long long M = 1;
    for (int mod : m) M *= mod;
    long long res = 0;
    for (int i = 0; i < a.size(); i++) {
        long long Mi = M / m[i];
        long long inv = modinv(Mi % m[i], m[i]);
        res = (res + 1LL * a[i] * Mi % M * inv % M) % M;
    }
    return (res % M + M) % M;
}
```

C. Example

Solve:

```
x   2 (mod 3)
x   3 (mod 5)
x   2 (mod 7)
```

Solution: (x = 23) (mod 105)

Check:

- (23 % 3 = 2)- (23 % 5 = 3)- (23 % 7 = 2)

4. Tiny Code

GCD

```
int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }
```

Modular Power

```
modpow(a, b, m)
```

Modular Inverse

```
modinv(a, m)
```

CRT

```
crt(a[], m[])
```

5. Summary

Concept	Formula	Purpose
GCD	$\gcd(a, b) = \gcd(b, a \bmod b)$	Simplify ratios
Extended GCD	$ax + by = \gcd(a, b)$	Find modular inverse
Modular Inverse	$a^{-1} \equiv a^{m-2} \pmod{m}$	Solve modular equations
Modular Exp	$a^b \bmod m$	Fast exponentiation
CRT	Combine congruences	Multi-mod system

Why It Matters

Number theory lets algorithms speak the language of integers , turning huge computations into modular games. From hashing to RSA, from combinatorics to cryptography, it's everywhere.

> "When numbers wrap around, math becomes modular , and algorithms become magical."

Try It Yourself

1. Compute gcd(48, 180).
2. Find inverse of 7 mod 13.
3. Solve $x \equiv 1 \pmod{2}, x \equiv 2 \pmod{3}, x \equiv 3 \pmod{5}$.
4. Implement modular division a/b mod m.
5. Use modpow to compute 3^{200} mod 13.

These basics unlock higher algorithms in cryptography, combinatorics, and beyond.

52. Primality and Factorization (Miller-Rabin, Pollard Rho)

Primality and factorization are core to number theory, cryptography, and competitive programming. Many modern systems (RSA, ECC) rely on the hardness of factoring large numbers. Here, we learn how to test if a number is prime and break it into factors efficiently.

We'll cover:

- Trial Division
- Sieve of Eratosthenes (for precomputation)
- Probabilistic Primality Test (Miller-Rabin)
- Integer Factorization (Pollard Rho)

1. Trial Division

The simplest way to test primality is by dividing by all integers up to √n.

```
bool is_prime(long long n) {
    if (n < 2) return false;
    if (n % 2 == 0) return n == 2;
    for (long long i = 3; i * i <= n; i += 2)
        if (n % i == 0) return false;
    return true;
}
```

Time Complexity: (O$\sqrt{n}$) Good for $n \leq 10^6$, impractical for large (n).

2. Sieve of Eratosthenes

For checking many numbers at once, use a sieve.

Idea: Mark all multiples of each prime starting from 2.

```
vector<bool> sieve(int n) {
    vector<bool> is_prime(n+1, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i * i <= n; i++)
        if (is_prime[i])
            for (int j = i * i; j <= n; j += i)
                is_prime[j] = false;
    return is_prime;
}
```

Time Complexity: ($O n \log \log n$)

Useful for generating primes up to 10^7.

3. Modular Multiplication

Before we do probabilistic tests or factorization, we need safe modular multiplication for large numbers.

```
long long modmul(long long a, long long b, long long m) {
    __int128 res = (__int128)a * b % m;
    return (long long)res;
}
```

Avoid overflow for $n \approx 10^{18}$.

4. Miller-Rabin Primality Test

A probabilistic test that can check if (n) is prime or composite in ($O k \log^3 n$).

Idea: For a prime (n):

$$a^{n-1} \equiv 1 \pmod n$$

But for composites, most (a) fail this.

We write $n - 1 = 2^s \cdot d$, (d) odd.

For each base (a):

- Compute $x = a^d \bmod n$- If (x = 1) or (x = n - 1), pass- Else, square (s-1) times- If none equal (n - 1), composite

```
bool miller_rabin(long long n) {
    if (n < 2) return false;
    for (long long p : {2,3,5,7,11,13,17,19,23,29,31,37})
        if (n % p == 0) return n == p;
    long long d = n - 1, s = 0;
    while ((d & 1) == 0) d >>= 1, s++;
    auto modpow = [&](long long a, long long b) {
        long long r = 1;
        while (b) {
            if (b & 1) r = modmul(r, a, n);
            a = modmul(a, a, n);
            b >>= 1;
        }
        return r;
    };
    for (long long a : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) {
        if (a % n == 0) continue;
        long long x = modpow(a, d);
        if (x == 1 || x == n - 1) continue;
        bool composite = true;
        for (int r = 1; r < s; r++) {
            x = modmul(x, x, n);
            if (x == n - 1) {
                composite = false;
                break;
            }
        }
        if (composite) return false;
    }
    return true;
}
```

Deterministic for:

- $n < 2^{64}$ with bases above. Complexity: ($Ok \log^3 n$)

5. Pollard Rho Factorization

Efficient for finding nontrivial factors of large composites. Based on Floyd's cycle detection (Tortoise and Hare).

Idea: Define a pseudo-random function:

$$f(x) = (x^2 + c) \bmod n$$

Then find $\gcd(|x - y|, n)$ where x, y move at different speeds.

```
long long pollard_rho(long long n) {
    if (n % 2 == 0) return 2;
    auto f = [&](long long x, long long c) {
        return (modmul(x, x, n) + c) % n;
    };
    while (true) {
        long long x = rand() % (n - 2) + 2;
        long long y = x;
        long long c = rand() % (n - 1) + 1;
        long long d = 1;
        while (d == 1) {
            x = f(x, c);
            y = f(f(y, c), c);
            d = gcd(abs(x - y), n);
        }
        if (d != n) return d;
    }
}
```

Use:

1. Check if (n) is prime (Miller-Rabin)
2. If not, find a factor using Pollard Rho
3. Recurse on factors

Complexity: ~ ($On^{1/4}$) average

6. Example

Factorize (n = 8051):

1. Miller-Rabin → composite
2. Pollard Rho → factor 83
3. (8051 / 83 = 97)
4. Both primes ($8051 = 83 \times 97$)

7. Tiny Code

```
void factor(long long n, vector<long long> &f) {
    if (n == 1) return;
    if (miller_rabin(n)) {
        f.push_back(n);
        return;
    }
    long long d = pollard_rho(n);
    factor(d, f);
    factor(n / d, f);
}
```

Call `factor(n, f)` to get prime factors.

8. Summary

Algorithm	Purpose	Complexity	Type
Trial Division	Small primes	(O$\sqrt{n}$)	Deterministic
Sieve	Precompute primes	(O$n \log \log n$)	Deterministic
Miller-Rabin	Primality test	(O$k \log^3 n$)	Probabilistic
Pollard Rho	Factorization	(O$n^{1/4}$)	Probabilistic

Why It Matters

Modern security, number theory problems, and many algorithmic puzzles depend on knowing when a number is prime and factoring it quickly when it isn't. These tools are the entry point to RSA, modular combinatorics, and advanced cryptography.

Try It Yourself

1. Check if 97 is prime using trial division and Miller-Rabin.
2. Factorize 5959 (should yield 59 $\times$ 101).
3. Generate all primes 100 using a sieve.
4. Write a recursive factorizer using Pollard Rho + Miller-Rabin.
5. Measure performance difference between $\sqrt{n}$ trial and Pollard Rho for $n \approx 10^{12}$.

These techniques make huge numbers approachable , one factor at a time.

53. Combinatorics (Permutations, Combinations, Subsets)

Combinatorics is the art of counting structures , how many ways can we arrange, select, or group things? In algorithms, it's everywhere: DP transitions, counting paths, bitmask enumeration, and probabilistic reasoning. Here we'll build a toolkit for computing factorials, nCr, nPr, and subset counts, both exactly and under a modulus.

1. Factorials and Precomputation

Most combinatorial formulas rely on factorials:

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

We can precompute them modulo (m) (often $10^9 + 7$) for efficiency.

```cpp
const int MOD = 1e9 + 7;
const int MAXN = 1e6;
long long fact[MAXN + 1], invfact[MAXN + 1];

long long modpow(long long a, long long b) {
    long long res = 1;
    while (b > 0) {
        if (b & 1) res = res * a % MOD;
        a = a * a % MOD;
        b >>= 1;
    }
    return res;
}

void init_factorials() {
    fact[0] = 1;
    for (int i = 1; i <= MAXN; i++)
        fact[i] = fact[i - 1] * i % MOD;
    invfact[MAXN] = modpow(fact[MAXN], MOD - 2);
    for (int i = MAXN - 1; i >= 0; i--)
        invfact[i] = invfact[i + 1] * (i + 1) % MOD;
}
```

Now you can compute (nCr) and (nPr) in (O(1)) time.

2. Combinations (nCr)

The number of ways to choose r items from (n) items:

$$C(n, r) = \frac{n!}{r!(n-r)!}$$

```
long long nCr(int n, int r) {
    if (r < 0 || r > n) return 0;
    return fact[n] * invfact[r] % MOD * invfact[n - r] % MOD;
}
```

Properties:

- $(C(n,0) = 1)$, $(C(n,n) = 1)$
- $C(n,r) = C(n,n-r)$
- Pascal's Rule: $C(n,r) = C(n-1,r-1) + C(n-1,r)$

Example

(C(5, 2) = 10) There are 10 ways to pick 2 elements from a 5-element set.

3. Permutations (nPr)

Number of ways to arrange r elements chosen from (n):

$$P(n,r) = \frac{n!}{(n-r)!}$$

```
long long nPr(int n, int r) {
    if (r < 0 || r > n) return 0;
    return fact[n] * invfact[n - r] % MOD;
}
```

Example

(P(5, 2) = 20) Choosing 2 out of 5 elements and arranging them yields 20 orders.

4. Subsets and Power Set

Each element has 2 choices: include or exclude. Hence, number of subsets:

$$2^n$$

```
long long subsets_count(int n) {
    return modpow(2, n);
}
```

Enumerating subsets using bitmasks:

```
for (int mask = 0; mask < (1 << n); mask++) {
    for (int i = 0; i < n; i++)
        if (mask & (1 << i))
            ; // include element i
}
```

Total: 2^n subsets, ($On2^n$) time to enumerate.

5. Multisets and Repetition

Number of ways to choose (r) items from (n) with repetition:

$$C(n + r - 1, r)$$

For example, number of ways to give 5 candies to 3 kids (each can get 0): (C(3+5-1, 5) = C(7,5) = 21)

6. Modular Combinatorics

When working modulo (p): - Use modular inverse for division. - $C(n, r) \bmod p = fact[n] \cdot invfact[r] \cdot invfact[n - r] \bmod p$

When $n \geq p$, use Lucas' Theorem:

$$C(n, r) \bmod p = C(n/p, r/p) \cdot C(n$$

7. Stirling and Bell Numbers (Advanced)

- Stirling Numbers of 2nd Kind: ways to partition (n) items into (k) non-empty subsets

$$S(n, k) = k \cdot S(n - 1, k) + S(n - 1, k - 1)$$

- Bell Numbers: total number of partitions

$$B(n) = \sum_{k=0}^{n} S(n, k)$$

Used in set partition and grouping problems.

8. Tiny Code

```
init_factorials();
printf("%lld\n", nCr(10, 3));  // 120
printf("%lld\n", nPr(10, 3));  // 720
printf("%lld\n", subsets_count(5)); // 32
```

9. Summary

Concept	Formula	Meaning	Example
Factorial	$n!$	All arrangements	$5! = 120$
Combina-tion	$C(n, r) = \frac{n!}{r!(n-r)!}$	Choose	$C(5, 2) = 10$
Permutation	$P(n, r) = \frac{n!}{(n-r)!}$	Arrange	$P(5, 2) = 20$
Subsets	2^n	All combinations	$2^3 = 8$
Multisets	$C(n + r - 1, r)$	Repetition allowed	$C(4, 2) = 6$

Why It Matters

Combinatorics underlies probability, DP counting, and modular problems. You can't master competitive programming or algorithm design without counting possibilities correctly. It teaches how structure emerges from choice , and how to count it efficiently.

Try It Yourself

1. Compute $C(1000, 500) \bmod (10^9 + 7)$.
2. Count the number of 5-bit subsets with exactly 3 bits on, i.e. $C(5, 3)$.
3. Write a loop to print all subsets of `{a, b, c, d}`.
4. Use Lucas' theorem for $C(10^6, 1000) \bmod 13$.
5. Implement Stirling recursion and print $S(5, 2)$.

Every algorithmic counting trick , from Pascal's triangle to binomial theorem , starts right here.

54. Probability and Randomized Algorithms

Probability introduces controlled randomness into computation. Instead of deterministic steps, randomized algorithms use random choices to achieve speed, simplicity, or robustness. This section bridges probability theory and algorithm design , teaching how to model, analyze, and exploit randomness.

We'll cover:

- Probability Basics
- Expected Value
- Monte Carlo and Las Vegas Algorithms
- Randomized Data Structures and Algorithms

1. Probability Basics

Every event has a probability between 0 and 1.
If a sample space has n equally likely outcomes and k of them satisfy a condition, then

$$P(E) = \frac{k}{n}$$

Examples

- Rolling a fair die: $P(\text{even}) = \frac{3}{6} = \frac{1}{2}$
- Drawing an ace from a deck: $P(\text{ace}) = \frac{4}{52} = \frac{1}{13}$

Key Rules

- Complement: $P(\bar{E}) = 1 - P(E)$
- Addition: $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- Multiplication: $P(A \cap B) = P(A) \cdot P(B \mid A)$

2. Expected Value

The expected value is the weighted average of outcomes.

$$E[X] = \sum_i P(x_i) \cdot x_i$$

Example: Expected value of a die:

$$E[X] = \frac{1+2+3+4+5+6}{6} = 3.5$$

Properties:

- Linearity: $E[aX + bY] = aE[X] + bE[Y]$

- Useful for average-case analysis

In algorithms:

- Expected number of comparisons in QuickSort is $O(n \log n)$
- Expected time for hash table lookup is $O(1)$

3. Monte Carlo vs Las Vegas

Randomized algorithms are broadly two types:

Type	Output	Runtime	Example
Monte Carlo	May be wrong (probabilistically)	Fixed	Miller-Rabin Primality
Las Vegas	Always correct	Random runtime	Randomized QuickSort

Monte Carlo:

- Faster, approximate
- You can control error probability
- E.g. primality test returns "probably prime"

Las Vegas:

- Output guaranteed correct
- Runtime varies by luck
- E.g. QuickSort with random pivot

4. Randomization in Algorithms

Randomization helps break worst-case patterns.

A. Randomized QuickSort

Pick a random pivot instead of first element. Expected time becomes ($On \log n$) regardless of input order.

```
int partition(int a[], int l, int r) {
    int pivot = a[l + rand() % (r - l + 1)];
    // move pivot to end, then normal partition
}
```

This avoids adversarial inputs like sorted arrays.

B. Randomized Hashing

Hash collisions can be exploited by adversaries. Using random coefficients in hash functions makes attacks infeasible.

```
long long hash(long long x, long long a, long long b, long long p) {
    return (a * x + b) % p;
}
```

Pick random (a, b) for robustness.

C. Randomized Data Structures

1. Skip List: uses random levels for nodes Expected ($O \log n$) search/insert/delete
2. Treap: randomized heap priority + BST order Maintains balance in expectation

```
struct Node {
    int key, priority;
    Node *l, *r;
};
```

Randomized balancing gives good average performance without rotation logic.

D. Random Sampling

Pick random elements efficiently:

- Reservoir Sampling: sample (k) items from a stream of unknown size- Shuffle: Fisher-Yates Algorithm

```
for (int i = n - 1; i > 0; i--) {
    int j = rand() % (i + 1);
    swap(a[i], a[j]);
}
```

5. Probabilistic Guarantees

Randomized algorithms often use Chernoff bounds and Markov's inequality to bound errors:

- Markov: $P(X \geq kE[X]) \leq \frac{1}{k}$
- Chebyshev: $P(|X - E[X]| \geq c\sigma) \leq \frac{1}{c^2}$
- Chernoff: Exponentially small tail bounds

These ensure "with high probability" $(1 - \frac{1}{n^c})$ guarantees.

6. Tiny Code

Randomized QuickSort:

```
int partition(int arr[], int low, int high) {
    int pivotIdx = low + rand() % (high - low + 1);
    swap(arr[pivotIdx], arr[high]);
    int pivot = arr[high], i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) swap(arr[i++], arr[j]);
    }
    swap(arr[i], arr[high]);
    return i;
}

void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}
```

7. Summary

Concept	Key Idea	Use Case
Expected Value	Weighted average outcome	Analyze average case
Monte Carlo	Probabilistic correctness	Primality test
Las Vegas	Probabilistic runtime	QuickSort
Random Pivot	Break worst-case	Sorting
Skip List / Treap	Random balancing	Data Structures

Concept	Key Idea	Use Case
Reservoir Sampling	Stream selection	Large data

Why It Matters

Randomization is not “luck” , it’s a design principle. It transforms rigid algorithms into adaptive, robust systems. In complexity theory, randomness helps achieve bounds impossible deterministically.

> “A bit of randomness turns worst cases into best friends.”

Try It Yourself

1. Simulate rolling two dice and compute expected sum.
2. Implement randomized QuickSort and measure average runtime.
3. Write a Monte Carlo primality checker.
4. Create a random hash function for integers.
5. Implement reservoir sampling for a large input stream.

These experiments show how uncertainty can become a powerful ally in algorithm design.

55. Sieve Methods and Modular Math

Sieve methods are essential tools in number theory for generating prime numbers, prime factors, and function values (,) efficiently. Combined with modular arithmetic, they form the backbone of algorithms in cryptography, combinatorics, and competitive programming.

This section introduces:

- Sieve of Eratosthenes- Optimized Linear Sieve- Sieve for Smallest Prime Factor (SPF)- Euler’s Totient Function ( )- Modular Applications

1. The Sieve of Eratosthenes

The classic algorithm to find all primes (n).

Idea: Start from 2, mark all multiples as composite. Continue to √n.

```cpp
vector<int> sieve(int n) {
    vector<int> primes;
    vector<bool> is_prime(n + 1, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i * i <= n; i++)
        if (is_prime[i])
            for (int j = i * i; j <= n; j += i)
                is_prime[j] = false;
    for (int i = 2; i <= n; i++)
        if (is_prime[i]) primes.push_back(i);
    return primes;
}
```

Time Complexity: ($On \log \log n$)

Space: (O(n))

Example: Primes up to 20 → 2, 3, 5, 7, 11, 13, 17, 19

2. Linear Sieve (O(n))

Unlike the basic sieve, each number is marked exactly once by its smallest prime factor (SPF).

Idea:

- For each prime (p), mark $p \times i$ only once.- Use `spf[i]` to store smallest prime factor.

```cpp
const int N = 1e6;
int spf[N + 1];
vector<int> primes;

void linear_sieve() {
    for (int i = 2; i <= N; i++) {
        if (!spf[i]) {
            spf[i] = i;
            primes.push_back(i);
        }
        for (int p : primes) {
            if (p > spf[i] || 1LL * i * p > N) break;
            spf[i * p] = p;
        }
    }
}
```

Benefits:

- Get primes, SPF, and factorizations in (O(n)).- Ideal for problems needing many factorizations.

3. Smallest Prime Factor (SPF) Table

With `spf[]`, factorization becomes (O $\log n$).

```
vector<int> factorize(int x) {
    vector<int> f;
    while (x != 1) {
        f.push_back(spf[x]);
        x /= spf[x];
    }
    return f;
}
```

Example: spf[12] = 2 → factors = [2, 2, 3]

4. Euler's Totient Function ((n))

The number of integers   (n) that are coprime with (n).

Formula:

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

Properties:

- $\varphi(p) = p - 1$ if p is prime
- Multiplicative: if $\gcd(a, b) = 1$, then $\varphi(ab) = \varphi(a)\varphi(b)$

Implementation (Linear Sieve):

```
const int N = 1e6;
int phi[N + 1];
bool is_comp[N + 1];
vector<int> primes;

void phi_sieve() {
    phi[1] = 1;
    for (int i = 2; i <= N; i++) {
        if (!is_comp[i]) {
            primes.push_back(i);
            phi[i] = i - 1;
        }
        for (int p : primes) {
            if (1LL * i * p > N) break;
            is_comp[i * p] = true;
            if (i % p == 0) {
                phi[i * p] = phi[i] * p;
                break;
            } else {
                phi[i * p] = phi[i] * (p - 1);
            }
        }
    }
}
```

Example:

- $\varphi(6) = 6(1 - \frac{1}{2})(1 - \frac{1}{3}) = 2$
- Numbers coprime with 6: 1, 5

5. Modular Math Applications

Once we have primes and totients, we can do many modular computations.

A. Fermat's Little Theorem

If (p) is prime,

$$a^{p-1} \equiv 1 \pmod{p}$$

Hence,

$$a^{-1} \equiv a^{p-2} \pmod{p}$$

Used in: modular inverses, combinatorics.

B. Euler's Theorem

If $\gcd(a,n)=1$, then

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

Generalizes Fermat's theorem to composite moduli.

C. Modular Exponentiation with Totient Reduction

For very large powers:

$$a^b \bmod n = a^{b \bmod \varphi(n)} \bmod n$$

(when a and n are coprime)

6. Tiny Code

Primes up to n:

```
auto primes = sieve(100);
```

Totients up to n:

```
phi_sieve();
cout << phi[10]; // 4
```

Factorization:

```
auto f = factorize(60); // [2, 2, 3, 5]
```

Concept	Description	Time	Use

7. Summary

Concept	Description	Time	Use
Eratosthenes	Mark multiples	(O$n \log \log n$)	Simple prime gen
Linear Sieve	Mark once	(O(n))	Prime + SPF
SPF Table	Smallest prime factor	(O(1)) query	Fast factorization
(n)	Coprime count	(O(n))	Modular exponent
Fermat / Euler	Inverses, reduction	(O$\log n$)	Modular arithmetic

Why It Matters

Sieve methods are the fastest way to preprocess arithmetic information. They unlock efficient solutions to problems involving primes, divisors, modular equations, and cryptography.

> "Before you can reason about numbers, you must first sieve them clean."

Try It Yourself

1. Generate all primes $\leq 10^6$ using a linear sieve.
2. Factorize 840 using the SPF array.
3. Compute $\varphi(n)$ for $n = 1..20$.
4. Verify $a^{\varphi(n)} \equiv 1 \pmod{n}$ for $a = 3$, $n = 10$.
5. Solve $a^b \bmod n$ with b very large using $\varphi(n)$.

Sieve once, and modular math becomes effortless forever after.

56. Linear Algebra (Gaussian Elimination, LU, SVD)

Linear algebra gives algorithms their mathematical backbone. From solving equations to powering ML models, it's the hidden engine behind optimization, geometry, and numerical computation.

In this section, we'll focus on the algorithmic toolkit:

- Gaussian Elimination (solve systems, invert matrices)
- LU Decomposition (efficient repeated solving)
- SVD (Singular Value Decomposition) overview

You'll see how algebra becomes code , step by step.

1. Systems of Linear Equations

We want to solve:

$$A \cdot x = b$$

where (A) is an $n \times n$ matrix, and (x, b) are vectors.

For example:

$$\{2x + 3y = 8\ x + 2y = 5$$

The solution is the intersection of two lines. In general, $A^{-1}b$ gives (x), but we usually solve it more directly using Gaussian elimination.

2. Gaussian Elimination (Row Reduction)

Idea: Transform ([A|b]) (augmented matrix) into upper-triangular form, then back-substitute.

Steps:

1. For each row, select a pivot (non-zero leading element).
2. Eliminate below it using row operations.
3. After all pivots, back-substitute to get the solution.

A. Implementation (C)

```
const double EPS = 1e-9;

vector<double> gauss(vector<vector<double>> A, vector<double> b) {
    int n = A.size();
    for (int i = 0; i < n; i++) {
        // 1. Find pivot
        int pivot = i;
        for (int j = i + 1; j < n; j++)
            if (fabs(A[j][i]) > fabs(A[pivot][i]))
                pivot = j;
        swap(A[i], A[pivot]);
        swap(b[i], b[pivot]);
        // 2. Normalize pivot row
        double div = A[i][i];
        if (fabs(div) < EPS) continue;
        for (int k = i; k < n; k++) A[i][k] /= div;
        b[i] /= div;

        // 3. Eliminate below
        for (int j = i + 1; j < n; j++) {
            double factor = A[j][i];
            for (int k = i; k < n; k++) A[j][k] -= factor * A[i][k];
            b[j] -= factor * b[i];
        }
    }

    // 4. Back substitution
    vector<double> x(n);
    for (int i = n - 1; i >= 0; i--) {
        x[i] = b[i];
        for (int j = i + 1; j < n; j++)
            x[i] -= A[i][j] * x[j];
    }
    return x;
}
```

Time complexity: (On^3)

B. Example

Solve:

$$\{2x + 3y = 8 \ x + 2y = 5$$

Augmented matrix:

$$[2 \quad 3 \quad | \quad 8 \, 1 \quad 2 \quad | \quad 5]$$

Reduce:

- Row2 ← Row2 − ½ Row1 → $[1, 2|5]$ → $[0, 0.5|1]$- Back substitute → (y = 2, x = 1)

3. LU Decomposition

LU factorization expresses:

$$A = L \cdot U$$

where (L) is lower-triangular (1s on diagonal), (U) is upper-triangular.

This allows solving (A x = b) in two triangular solves:

1. Solve (L y = b)
2. Solve (U x = y)

Efficient when solving for multiple b's (same A).

A. Decomposition Algorithm

```
void lu_decompose(vector<vector<double>>& A, vector<vector<double>>& L, vector<vector<double
    int n = A.size();
    L.assign(n, vector<double>(n, 0));
    U.assign(n, vector<double>(n, 0));

    for (int i = 0; i < n; i++) {
        // Upper
        for (int k = i; k < n; k++) {
            double sum = 0;
            for (int j = 0; j < i; j++)
                sum += L[i][j] * U[j][k];
            U[i][k] = A[i][k] - sum;
        }
        // Lower
        for (int k = i; k < n; k++) {
            if (i == k) L[i][i] = 1;
            else {
                double sum = 0;
                for (int j = 0; j < i; j++)
                    sum += L[k][j] * U[j][i];
                L[k][i] = (A[k][i] - sum) / U[i][i];
            }
        }
    }
}
```

Solve with forward + backward substitution.

4. Singular Value Decomposition (SVD)

SVD generalizes diagonalization for non-square matrices:

$$A = U\Sigma V^T$$

Where:

- (U): left singular vectors (orthogonal)- Σ: diagonal of singular values- V^T: right singular vectors Applications:
- Data compression (PCA)- Noise reduction- Rank estimation- Pseudoinverse $A^+ = V\Sigma^{-1}U^T$ In practice, use libraries (e.g. LAPACK, Eigen).

5. Numerical Stability and Pivoting

In floating-point math:

- Always pick largest pivot (partial pivoting)- Avoid dividing by small numbers- Use EPS = 1e-9 threshold Small numerical errors can amplify quickly , stability is key.

6. Tiny Code

```
vector<vector<double>> A = {{2, 3}, {1, 2}};
vector<double> b = {8, 5};
auto x = gauss(A, b);
// Output: x = [1, 2]
```

7. Summary

Algorithm	Purpose	Complexity	Notes
Gaussian Elimination	Solve Ax=b	(On^3)	Direct method
LU Decomposition	Repeated solves	(On^3)	Triangular factorization
SVD	General decomposition	(On^3)	Robust, versatile

Why It Matters

Linear algebra is the language of algorithms , it solves equations, optimizes functions, and projects data. Whether building solvers or neural networks, these methods are your foundation.

"Every algorithm lives in a vector space , it just needs a basis to express itself."

Try It Yourself

1. Solve a 3×3 linear system with Gaussian elimination.
2. Implement LU decomposition and test $L \cdot U = A$.
3. Use LU to solve multiple (b) vectors.
4. Explore SVD using a math library; compute singular values of a 2×2 matrix.
5. Compare results between naive and pivoted elimination for unstable systems.

Start with row operations , and you'll see how geometry and algebra merge into code.

57. FFT and NTT (Fast Transforms)

The Fast Fourier Transform (FFT) is one of the most beautiful and practical algorithms ever invented. It converts data between time (or coefficient) domain and frequency (or point) domain efficiently. The Number Theoretic Transform (NTT) is its modular counterpart for integer arithmetic , ideal for polynomial multiplication under a modulus.

This section covers:

- Why we need transforms- Discrete Fourier Transform (DFT)- Cooley-Tukey FFT (complex numbers)- NTT (modular version)- Applications (polynomial multiplication, convolution)

1. Motivation

Suppose you want to multiply two polynomials:

$$A(x) = a_0 + a_1 x + a_2 x^2$$

$$B(x) = b_0 + b_1 x + b_2 x^2$$

Their product has coefficients:

$$c_k = \sum_{i+j=k} a_i \cdot b_j$$

This is convolution:

$$C = A * B$$

Naively, this takes (On^2). FFT reduces it to ($On \log n$).

2. Discrete Fourier Transform (DFT)

The DFT maps coefficients $a_0, a_1, \ldots, a_{n-1}$ to evaluations at (n)-th roots of unity:

$$A_k = \sum_{j=0}^{n-1} a_j \cdot e^{-2\pi i \cdot jk/n}$$

and the inverse transform recovers a_j from A_k.

3. Cooley-Tukey FFT

Key idea: recursively split the sum into even and odd parts:

$$A_k = A_{even}(w_n^2) + w_n^k \cdot A_{odd}(w_n^2)$$

Where $w_n = e^{-2\pi i/n}$ is an (n)-th root of unity.

Implementation (C++)

```
#include <complex>
#include <vector>
#include <cmath>
using namespace std;

using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd> &a, bool invert) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1) j ^= bit;
        j ^= bit;
        if (i < j) swap(a[i], a[j]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i + j], v = a[i + j + len / 2] * w;
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
                w *= wlen;
            }
        }
    }

    if (invert) {
        for (cd &x : a) x /= n;
    }
}
```

Polynomial Multiplication with FFT

```
vector<long long> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < (int)a.size() + (int)b.size()) n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++) fa[i] *= fb[i];
    fft(fa, true);

    vector<long long> result(n);
    for (int i = 0; i < n; i++)
        result[i] = llround(fa[i].real());
    return result;
}
```

Complexity: ($On\log n$)

4. Number Theoretic Transform (NTT)

FFT uses complex numbers , NTT uses modular arithmetic with roots of unity mod p. We need a prime (p) such that:

$$p = c \cdot 2^k + 1$$

so a primitive root (g) exists.

Popular choices:

- (p = 998244353, g = 3)- (p = 7340033, g = 3)

Implementation (NTT)

```
const int MOD = 998244353;
const int G = 3;

int modpow(int a, int b) {
    long long res = 1;
    while (b) {
        if (b & 1) res = res * a % MOD;
        a = 1LL * a * a % MOD;
        b >>= 1;
    }
    return res;
}

void ntt(vector<int> &a, bool invert) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1) j ^= bit;
        j ^= bit;
        if (i < j) swap(a[i], a[j]);
    }
    for (int len = 2; len <= n; len <<= 1) {
        int wlen = modpow(G, (MOD - 1) / len);
        if (invert) wlen = modpow(wlen, MOD - 2);
        for (int i = 0; i < n; i += len) {
            long long w = 1;
            for (int j = 0; j < len / 2; j++) {
                int u = a[i + j];
                int v = (int)(a[i + j + len / 2] * w % MOD);
                a[i + j] = u + v < MOD ? u + v : u + v - MOD;
                a[i + j + len / 2] = u - v >= 0 ? u - v : u - v + MOD;
                w = w * wlen % MOD;
            }
        }
    }
    if (invert) {
        int inv_n = modpow(n, MOD - 2);
        for (int &x : a) x = 1LL * x * inv_n % MOD;
    }
}
```

5. Applications

1. Polynomial Multiplication: $(O n \log n)$
2. Convolution: digital signal processing
3. Big Integer Multiplication (Karatsuba, FFT)
4. Subset Convolution and combinatorial transforms
5. Number-theoretic sums (NTT-friendly modulus)

6. Tiny Code

```
vector<int> A = {1, 2, 3};
vector<int> B = {4, 5, 6};
// Result = {4, 13, 28, 27, 18}
auto C = multiply(A, B);
```

7. Summary

Algorithm	Domain	Complexity	Type
DFT	Complex	$(O n^2)$	Naive
FFT	Complex	$(O n \log n)$	Recursive
NTT	Modular	$(O n \log n)$	Integer arithmetic

Why It Matters

FFT and NTT bring polynomial algebra to life. They turn slow convolutions into lightning-fast transforms. From multiplying huge integers to compressing signals, they're the ultimate divide-and-conquer on structure.

> "To multiply polynomials fast, you first turn them into music , then back again."

Try It Yourself

1. Multiply $(1 + 2x + 3x^2)$ and $(4 + 5x + 6x^2)$ using FFT.
2. Implement NTT over 998244353 and verify results mod p.
3. Compare (On^2) vs FFT performance for n = 1024.
4. Experiment with inverse FFT (invert = true).
5. Explore circular convolution for signal data.

Once you master FFT/NTT, you hold the power of speed in algebraic computation.

58. Numerical Methods (Newton, Simpson, Runge-Kutta)

Numerical methods let us approximate solutions when exact algebraic answers are hard or impossible. They are the foundation of scientific computing, simulation, and optimization , bridging the gap between continuous math and discrete computation.

In this section, we'll explore three classics:

- Newton-Raphson: root finding- Simpson's Rule: numerical integration- Runge-Kutta (RK4): solving differential equations These algorithms showcase how iteration, approximation, and convergence build powerful tools.

1. Newton-Raphson Method

Used to find a root of (f(x) = 0). Starting from a guess x_0, iteratively refine:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Convergence is quadratic if (f) is smooth and x_0 is close enough.

A. Example

Solve (f(x) = x^2 - 2 = 0) We know root = $\sqrt{2}$

Start $x_0 = 1$

Iter	x_n	(fx_n)	(f'x_n)	x_{n+1}
0	1.000	-1.000	2.000	1.500
1	1.500	0.250	3.000	1.417
2	1.417	0.006	2.834	1.414

Converged: 1.414 $\approx \sqrt{2}$

B. Implementation

```c
#include <math.h>
#include <stdio.h>

double f(double x) { return x * x - 2; }
double df(double x) { return 2 * x; }

double newton(double x0) {
    for (int i = 0; i < 20; i++) {
        double fx = f(x0);
        double dfx = df(x0);
        if (fabs(fx) < 1e-9) break;
        x0 = x0 - fx / dfx;
    }
    return x0;
}

int main() {
    printf("Root: %.6f\n", newton(1.0)); // 1.414214
}
```

Time Complexity: (O(k)) iterations, each (O(1))

2. Simpson's Rule (Numerical Integration)

When you can't integrate (f(x)) analytically, approximate the area under the curve.

Divide interval ([a, b]) into even (n) subintervals (width (h)).

$$I \approx \frac{h}{3}\Big(f(a) + 4\sum f(x_{odd}) + 2\sum f(x_{even}) + f(b)\Big)$$

A. Implementation

```c
#include <math.h>
#include <stdio.h>

double f(double x) { return x * x; } // integrate x^2

double simpson(double a, double b, int n) {
    double h = (b - a) / n;
    double s = f(a) + f(b);
    for (int i = 1; i < n; i++) {
        double x = a + i * h;
        s += f(x) * (i % 2 == 0 ? 2 : 4);
    }
    return s * h / 3;
}

int main() {
    printf("  ¹ x² dx   %.6f\n", simpson(0, 1, 100)); // ~0.333333
}
```

Accuracy: (Oh^4) Note: (n) must be even.

B. Example

$$\int_0^1 x^2 dx = \frac{1}{3}$$

With (n = 100), Simpson gives (0.333333).

3. Runge-Kutta (RK4)

Used to solve first-order ODEs:

$$y' = f(x, y), \quad y(x_0) = y_0$$

RK4 Formula:

$$k_1 = f(x_n, y_n)\ k_2 \quad = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1)\ k_3 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2)\ k_4 \quad = f(x_n + h, y_n + hk_3)\ y_{n+1} = y_n + \frac{h}{6}$$

Accuracy: (Oh^4)

A. Example

Solve (y' = x + y), (y(0) = 1), step (h = 0.1).

Each iteration refines (y) with weighted slope average.

B. Implementation

```c
#include <stdio.h>

double f(double x, double y) {
    return x + y;
}

double runge_kutta(double x0, double y0, double h, double xn) {
    double x = x0, y = y0;
    while (x < xn) {
        double k1 = f(x, y);
        double k2 = f(x + h / 2, y + h * k1 / 2);
        double k3 = f(x + h / 2, y + h * k2 / 2);
        double k4 = f(x + h, y + h * k3);
        y += h * (k1 + 2*k2 + 2*k3 + k4) / 6;
        x += h;
    }
    return y;
}
```

```
int main() {
    printf("y(0.1)   %.6f\n", runge_kutta(0, 1, 0.1, 0.1));
}
```

4. Tiny Code Summary

Method	Purpose	Formula	Accuracy	Type
Newton-Raphson	Root finding	$x_{n+1} = x_n - \frac{f}{f'}$	Quadratic	Iterative
Simpson's Rule	Integration	(h/3(...))	(Oh^4)	Deterministic
Runge-Kutta (RK4)	ODEs	Weighted slope avg	(Oh^4)	Iterative

5. Numerical Stability

- Small step (h): better accuracy, more cost- Large (h): faster, less stable- Always check convergence ($|x_{n+1} - x_n| < \varepsilon$)- Avoid division by small derivatives in Newton's method

Why It Matters

Numerical methods let computers simulate the continuous world. From physics to AI training, they solve what calculus cannot symbolically.

> "When equations won't talk, iterate , and they'll whisper their answers."

Try It Yourself

1. Use Newton's method for $\cos x - x = 0$.
2. Approximate $\int_0^{\pi/2} \sin x \, dx$ with Simpson's rule.
3. Solve $y' = y - x^2 + 1,\ y(0) = 0.5$ using RK4.
4. Compare RK4 with Euler's method for the same ODE.
5. Experiment with step sizes $h \in \{0.1, 0.01, 0.001\}$ and observe convergence.

Numerical thinking turns continuous problems into iterative algorithms , precise enough to power every simulation and solver you'll ever write.

59. Mathematical Optimization (Simplex, Gradient, Convex)

Mathematical optimization is about finding the best solution , smallest cost, largest profit, shortest path , under given constraints. It's the heart of machine learning, operations research, and engineering design.

In this section, we'll explore three pillars:

- Simplex Method , for linear programs
- Gradient Descent , for continuous optimization
- Convex Optimization , the theory ensuring global optima

1. What Is Optimization?

A general optimization problem looks like:

$$\min_x \; f(x)$$

subject to constraints:

$$g_i(x) \leq 0, \quad h_j(x) = 0$$

When (f) and g_i, h_j are linear, it's a Linear Program (LP). When (f) is differentiable, we can use gradients. When (f) is convex, every local minimum is global , the ideal world.

2. The Simplex Method (Linear Programming)

A linear program has the form:

$$\max \; c^T x$$

subject to

$$Ax \leq b, \quad x \geq 0$$

Geometrically, each constraint forms a half-space. The feasible region is a convex polytope, and the optimum lies at a vertex.

A. Example

Maximize (z = 3x + 2y) subject to

$$\left\{ 2x + y \leq 18 \;\; 2x + 3y \leq 42 \;\; x, y \geq 0 \right.$$

Solution: (x=9, y=8), (z=43)

B. Algorithm (Sketch)

1. Convert inequalities to equalities by adding slack variables.
2. Initialize at a vertex (basic feasible solution).
3. At each step:

    - Choose entering variable (most negative coefficient in objective). - Choose leaving variable (min ratio test). - Pivot to new vertex.4. Repeat until optimal.

C. Implementation (Simplified Pseudocode)

```
// Basic simplex-like outline
while (exists negative coefficient in objective row) {
    choose entering column j;
    choose leaving row i (min b[i]/a[i][j]);
    pivot(i, j);
}
```

Libraries (like `GLPK` or `Eigen`) handle full implementations.

Time Complexity: worst (O2^n), but fast in practice.

3. Gradient Descent

For differentiable (f(x)), we move opposite the gradient:

$$x_{k+1} = x_k - \eta \nabla f(x_k)$$

where η is the learning rate.

Intuition: (f(x)) points uphill, so step opposite it.

A. Example

Minimize (f(x) = (x-3)^2)

$$f'(x) = 2(x-3)$$

Start $x_0 = 0$, $\eta = 0.1$

Iter	x_k	(fx_k)	Gradient	New (x)
0	0	9	-6	0.6
1	0.6	5.76	-4.8	1.08
2	1.08	3.69	-3.84	1.46
…	→3	→0	→0	→3

Converges to (x = 3)

B. Implementation

```c
#include <math.h>
#include <stdio.h>

double f(double x) { return (x - 3) * (x - 3); }
double df(double x) { return 2 * (x - 3); }

double gradient_descent(double x0, double lr) {
    for (int i = 0; i < 100; i++) {
        double g = df(x0);
        if (fabs(g) < 1e-6) break;
        x0 -= lr * g;
    }
    return x0;
}

int main() {
    printf("Min at x = %.6f\n", gradient_descent(0, 0.1));
}
```

C. Variants

- Momentum: (v = v + 1 − β f(x))- Adam: adaptive learning rates- Stochastic Gradient Descent (SGD): random subset of data All used heavily in machine learning.

4. Convex Optimization

A function (f) is convex if:

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$$

This means any local minimum is global.

Examples:

- (f(x) = x^2) (convex)- (f(x) = x^3) (not convex) For convex functions with linear constraints, gradient-based methods always converge to the global optimum.

A. Checking Convexity

- 1D: (f' '(x) 0)- Multivariate: Hessian (^2 f(x)) is positive semidefinite

5. Applications

- Linear Programming (Simplex): logistics, scheduling- Quadratic Programming: portfolio optimization- Gradient Methods: ML, curve fitting- Convex Programs: control systems, regularization

6. Tiny Code

Simple gradient descent to minimize (f(x,y)=x^{2+y}2):

```
double f(double x, double y) { return x*x + y*y; }
void grad(double x, double y, double *gx, double *gy) {
    *gx = 2*x; *gy = 2*y;
}

void optimize() {
    double x=5, y=3, lr=0.1;
    for(int i=0; i<100; i++){
        double gx, gy;
        grad(x, y, &gx, &gy);
        x -= lr * gx;
        y -= lr * gy;
    }
    printf("Min at (%.3f, %.3f)\n", x, y);
}
```

7. Summary

Algorithm	Domain	Complexity	Notes
Simplex	Linear	Polynomial (average case)	LP solver
Gradient Descent	Continuous	$O(k)$	Needs step size
Convex Methods	Convex	$O(k \log \frac{1}{\varepsilon})$	Global optima guaranteed

Why It Matters

Optimization turns math into decisions. From fitting curves to planning resources, it formalizes trade-offs and efficiency. It's where computation meets purpose , finding the best in all possible worlds.

> "Every algorithm is, at heart, an optimizer , searching for something better."

Try It Yourself

1. Solve a linear program with 2 constraints manually via Simplex.
2. Implement gradient descent for $f(x) = (x-5)^2 + 2$.
3. Add momentum to your gradient descent loop.
4. Check convexity by plotting $f(x) = x^4 - 3x^2$.
5. Experiment with learning rates: too small leads to slow convergence; too large can diverge.

Mastering optimization means mastering how algorithms improve themselves , step by step, iteration by iteration.

60. Algebraic Tricks and Transform Techniques

In algorithm design, algebra isn't just theory , it's a toolbox for transforming problems. By expressing computations algebraically, we can simplify, accelerate, or generalize solutions. This section surveys common algebraic techniques that turn hard problems into manageable ones.

We'll explore:

- Algebraic identities and factorizations
- Generating functions and transforms
- Convolution tricks
- Polynomial methods and FFT applications
- Matrix and linear transforms for acceleration

1. Algebraic Identities

These let you rewrite or decompose expressions to reveal structure or reduce complexity.

Classic Forms:

- Difference of squares:
$$a^2 - b^2 = (a - b)(a + b)$$
- Sum of cubes:
$$a^3 + b^3 = (a + b)(a^2 - ab + b^2)$$
- Square of sum:
$$(a + b)^2 = a^2 + 2ab + b^2$$

Used in dynamic programming, geometry, and optimization when simplifying recurrence terms or constraints.

Example: Transforming $(x + y)^2$ lets you compute both $x^2 + y^2$ and cross terms efficiently.

2. Generating Functions

A generating function encodes a sequence $a_0, a_1, a_2, ...$ into a formal power series:

$$G(x) = a_0 + a_1x + a_2x^2 + ...$$

They turn recurrence relations and counting problems into algebraic equations.

Example: Fibonacci sequence

$$F(x) = F_0 + F_1x + F_2x^2 + ...$$

with recurrence $F_n = F_{n-1} + F_{n-2}$

Solve algebraically:

$$F(x) = \frac{x}{1 - x - x^2}$$

Applications: combinatorics, probability, counting partitions.

3. Convolution Tricks

Convolution arises in combining sequences:

$$(c_n) = (a * b) * n = \sum *i = 0^n a_i b_{n-i}$$

Naive computation: ($\mathrm{O}n^2$) Using Fast Fourier Transform (FFT): ($\mathrm{O}n \log n$)

Example: Polynomial multiplication Let

$$A(x) = a_0 + a_1 x + a_2 x^2, \quad B(x) = b_0 + b_1 x + b_2 x^2$$

Then (C(x) = A(x)B(x)) gives coefficients by convolution.

This trick is used in:

- Large integer multiplication- Pattern matching (cross-correlation)- Subset sum acceleration

4. Polynomial Methods

Many algorithmic problems can be represented as polynomials, where coefficients encode combinatorial structure.

A. Polynomial Interpolation

Given (n+1) points, there's a unique degree-(n) polynomial passing through them.

Used in error correction, FFT-based reconstruction, and number-theoretic transforms.

Lagrange Interpolation:

$$P(x) = \sum_i y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

B. Root Representation

Solve equations or check identities by working modulo a polynomial. Used in finite fields and coding theory (e.g., Reed-Solomon).

5. Transform Techniques

Transforms convert problems to simpler domains where operations become efficient.

Transform	Converts	Key Property	Used In
FFT / NTT	Time Frequency	Convolution $\rightarrow$ Multiplication	Signal, polynomial mult
Z-Transform	Sequence Function	Recurrence solving	DSP, control
Laplace Transform	Function Algebraic	Diff. eq. $\rightarrow$ Algebraic eq.	Continuous systems
Walsh-Hadamard Transform	Boolean vectors	XOR convolution	Subset sum, SOS DP

Example: Subset Convolution via FWT

For all subsets (S):

$$f'(S) = \sum_{T \subseteq S} f(T)$$

Use Fast Walsh-Hadamard Transform (FWHT) to compute in (O$n2^n$) instead of (O3^n).

6. Matrix Tricks

Matrix algebra enables transformations and compact formulations.

- Matrix exponentiation: solve recurrences in $O(\log n)$
- Diagonalization: $A = PDP^{-1}$, then $A^k = PD^kP^{-1}$
- Fast power: speeds up Fibonacci, linear recurrences, Markov chains

Example: Fibonacci

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

7. Tiny Code

Polynomial Multiplication via FFT (Pseudo-C):

```c
// Outline using complex FFT library
fft(A, false);
fft(B, false);
for (int i = 0; i < n; i++)
    C[i] = A[i] * B[i];
fft(C, true); // inverse
```

Matrix Power (Fibonacci):

```c
void matmul(long long A[2][2], long long B[2][2]) {
    long long C[2][2] = {{0}};
    for (int i=0;i<2;i++)
        for (int j=0;j<2;j++)
            for (int k=0;k<2;k++)
                C[i][j] += A[i][k]*B[k][j];
    memcpy(A, C, sizeof(C));
}

void matpow(long long A[2][2], int n) {
    long long R[2][2] = {{1,0},{0,1}};
    while(n){
        if(n&1) matmul(R,A);
        matmul(A,A);
        n>>=1;
    }
    memcpy(A, R, sizeof(R));
}
```

8. Summary

Technique	Purpose	Speedup
Algebraic Identities	Simplify expressions	Constant factor
Generating Functions	Solve recurrences	Conceptual
FFT / Convolution	Combine sequences fast	$(On^2 \to On \log n)$
Polynomial Interpolation	Reconstruction	$(On^2 \to On \log^2 n)$
Matrix Tricks	Accelerate recurrences	$(O(n) \to O\log n)$

Why It Matters

Algebra turns computation into structure. By rewriting problems in algebraic form, you reveal hidden symmetries, exploit fast transforms, and find elegant solutions. It's not magic , it's the math beneath performance.

> "The smartest code is often the one that solves itself on paper first."

Try It Yourself

1. Multiply two polynomials using FFT.
2. Represent Fibonacci as a matrix and compute F_{100}.
3. Use generating functions to count coin change ways.
4. Implement subset sum via Walsh-Hadamard transform.
5. Derive a recurrence and solve it algebraically.

Understanding algebraic tricks makes you not just a coder, but a mathematical engineer , bending structure to will.

Chapter 7. Strings and Text Algorithms

61. String Matching (KMP, Z, Rabin-Karp, Boyer-Moore)

String matching is one of the oldest and most fundamental problems in computer science: given a text (T) of length (n) and a pattern (P) of length (m), find all positions where (P) appears in (T).

This section walks you through both naive and efficient algorithms , from the straightforward brute-force method to elegant linear-time solutions like KMP and Z-algorithm, and clever heuristics like Boyer-Moore and Rabin-Karp.

1. Problem Setup

We're given:

- Text: $T = t_1 t_2 \dots t_n$- Pattern: $P = p_1 p_2 \dots p_m$ Goal: find all (i) such that

$$T[i \dots i + m - 1] = P[1 \dots m]$$

Naive solution: compare (P) with every substring of (T) Time complexity: (O(nm))

We'll now see how to reduce it to (O(n + m)) or close.

2. Knuth-Morris-Pratt (KMP)

KMP avoids rechecking characters by precomputing overlaps within the pattern.

It builds a prefix-function (also called failure function), which tells how much to shift when a mismatch happens.

A. Prefix Function

For each position (i), compute $\pi[i]$ = length of longest prefix that's also a suffix of (P[1..i]).

Example: Pattern `ababc`

i	P[i]	[i]
1	a	0
2	b	0
3	a	1
4	b	2
5	c	0

B. Search Phase

Use $\pi[]$ to skip mismatched prefixes in the text.

Time Complexity: (O(n + m)) Space: (O(m))

Tiny Code (C)

```
void compute_pi(char *p, int m, int pi[]) {
    pi[0] = 0;
    for (int i = 1, k = 0; i < m; i++) {
        while (k > 0 && p[k] != p[i]) k = pi[k-1];
        if (p[k] == p[i]) k++;
        pi[i] = k;
    }
}

void kmp_search(char *t, char *p) {
    int n = strlen(t), m = strlen(p);
    int pi[m]; compute_pi(p, m, pi);
    for (int i = 0, k = 0; i < n; i++) {
        while (k > 0 && p[k] != t[i]) k = pi[k-1];
        if (p[k] == t[i]) k++;
        if (k == m) {
            printf("Found at %d\n", i - m + 1);
            k = pi[k-1];
        }
    }
}
```

3. Z-Algorithm

Z-algorithm computes the Z-array,
where $Z[i]$ = length of the longest substring starting at i that matches the prefix of P.

To match P in T, build the string:

$$S = P + \# + T$$

Then every i where $Z[i] = |P|$ corresponds to a match.

Time: $O(n + m)$
Simple and elegant.

Example:

```
P = "aba", T = "ababa"
S = "aba#ababa"
Z = [0,0,1,0,3,0,1,0]
Match at index 0, 2
```

4. Rabin-Karp (Rolling Hash)

Instead of comparing strings character-by-character, compute a hash for each window in (T), and compare hashes.

$$h(s_1 s_2 \dots s_m) = (s_1 b^{m-1} + s_2 b^{m-2} + \dots + s_m) \bmod M$$

Use a rolling hash to update in (O(1)) per shift.

Time: average (O(n + m)), worst (O(nm)) Good for multiple pattern search.

Tiny Code (Rolling Hash)

```c
#define B 256
#define M 101

void rabin_karp(char *t, char *p) {
    int n = strlen(t), m = strlen(p);
    int h = 1, pHash = 0, tHash = 0;
    for (int i = 0; i < m-1; i++) h = (h*B) % M;
    for (int i = 0; i < m; i++) {
        pHash = (B*pHash + p[i]) % M;
        tHash = (B*tHash + t[i]) % M;
    }
    for (int i = 0; i <= n-m; i++) {
        if (pHash == tHash && strncmp(&t[i], p, m) == 0)
            printf("Found at %d\n", i);
        if (i < n-m)
            tHash = (B*(tHash - t[i]*h) + t[i+m]) % M;
        if (tHash < 0) tHash += M;
    }
}
```

5. Boyer-Moore (Heuristic Skipping)

Boyer-Moore compares from right to left and uses two heuristics:

1. Bad Character Rule When mismatch at (j), shift pattern so next occurrence of (T[i]) in (P) aligns.
2. Good Suffix Rule Shift pattern so a suffix of matched portion aligns with another occurrence.

Time: (O(n/m)) on average Practical and fast, especially for English text.

6. Summary

Algorithm	Time	Space	Idea	Best For
Naive	(O(nm))	(O(1))	Direct compare	Simple cases
KMP	(O(n+m))	(O(m))	Prefix overlap	General use
Z	(O(n+m))	(O(n+m))	Prefix matching	Pattern prep
Rabin-Karp	(O(n+m)) avg	(O(1))	Hashing	Multi-pattern
Boyer-Moore	(O(n/m)) avg	$(Om + \sigma)$	Right-to-left skip	Long texts

Why It Matters

String matching powers text editors, DNA search, spam filters, and search engines. These algorithms show how structure and clever preprocessing turn brute force into elegance.

> "To find is human, to match efficiently is divine."

Try It Yourself

1. Implement KMP and print all matches in a sentence.
2. Use Rabin-Karp to find multiple keywords.
3. Compare running times on large text files.
4. Modify KMP for case-insensitive matching.
5. Visualize prefix function computation step-by-step.

By mastering these, you'll wield the foundation of pattern discovery , the art of finding order in streams of symbols.

62. Multi-Pattern Search (Aho-Corasick)

So far, we've matched one pattern against a text. But what if we have many patterns , say, a dictionary of keywords , and we want to find all occurrences of all patterns in a single pass?

That's where the Aho-Corasick algorithm shines. It builds a trie with failure links, turning multiple patterns into one efficient automaton. Think of it as "KMP for many words at once."

1. Problem Setup

Given:

- A text (T) of length (n)- A set of patterns $P_1, P_2, \dots, P_k$ with total length $m = \sum |P_i|$

Goal: find all occurrences of every P_i in (T).

Naive solution: Run KMP for each pattern , (O(kn))

Better idea: Merge all patterns into a trie, and use failure links to transition on mismatches.

Aho-Corasick achieves O(n + m + z), where (z) = number of matches reported.

2. Trie Construction

Each pattern is inserted into a trie node-by-node.

Example Patterns:

```
he, she, his, hers
```

Trie:

```
(root)
   h   e*
       r   s*
   s   h   e*
   h   i   s*
```

Each node may mark an output (end of pattern).

3. Failure Links

Failure link of a node points to the longest proper suffix that's also a prefix in the trie.

These links let us "fall back" like KMP.

When mismatch happens, follow failure link to find next possible match.

Building Failure Links (BFS)

1. Root's failure = null
2. Children of root → failure = root
3. BFS over nodes:
   - For each edge ((u, c) → v): follow failure links from (u) until you find (f) with edge (c) then $v.\text{fail} = f.c$

Example

For "he", "she", "his", "hers":

- `fail("he") = root`- `fail("hers") = "rs"` path invalid → fallback to `"s"` if exists So failure links connect partial suffixes.

4. Matching Phase

Now we can process the text in one pass:

```
state = root
for each character c in text:
    while state has no child c and state != root:
        state = state.fail
    if state has child c:
        state = state.child[c]
    else:
        state = root
    if state.output:
        report matches at this position
```

Each transition costs O(1) amortized. No backtracking , fully linear time.

5. Example Walkthrough

Patterns: `he`, `she`, `his`, `hers` Text: `ahishers`

At each character:

```
a → root (no match)
h → go to h
i → go to hi
s → go to his → output "his"
h → fallback → h
e → he → output "he"
r → her → continue
s → hers → output "hers"
```

Outputs: `"his"`, `"he"`, `"hers"`

6. Tiny Code (C Implementation Sketch)

```c
#define ALPHA 26

typedef struct Node {
    struct Node *next[ALPHA];
    struct Node *fail;
    int out;
} Node;

Node* newNode() {
    Node *n = calloc(1, sizeof(Node));
    return n;
}

void insert(Node *root, char *p) {
    for (int i = 0; p[i]; i++) {
        int c = p[i] - 'a';
        if (!root->next[c]) root->next[c] = newNode();
        root = root->next[c];
    }
    root->out = 1;
}

void build_failures(Node *root) {
    Node *q[10000];
    int front=0, back=0;
    root->fail = root;
    q[back++] = root;
    while (front < back) {
        Node *u = q[front++];
        for (int c=0; c<ALPHA; c++) {
            Node *v = u->next[c];
            if (!v) continue;
            Node *f = u->fail;
            while (f != root && !f->next[c]) f = f->fail;
            if (f->next[c] && f->next[c] != v) v->fail = f->next[c];
            else v->fail = root;
            if (v->fail->out) v->out = 1;
            q[back++] = v;
        }
    }
}
```

7. Complexity

Phase	Time	Space
Trie Build	(O(m))	(O(m))
Failure Links	(O(m))	(O(m))
Search	(O(n + z))	(O(1))

Total: O(n + m + z)

8. Summary

Step	Purpose
Trie	Merge patterns
Fail Links	Handle mismatches
Outputs	Collect matches
BFS	Build efficiently
One Pass	Match all patterns

Why It Matters

Aho-Corasick is the core of:

- Spam filters- Intrusion detection (e.g., Snort IDS)- Keyword search in compilers- DNA sequence scanners It's a masterclass in blending automata theory with practical efficiency.

  "Why search one word at a time when your algorithm can read the whole dictionary?"

Try It Yourself

1. Build an automaton for words {"he", "she", "hers"} and trace it manually.
2. Modify code for uppercase letters.
3. Extend to report overlapping matches.
4. Measure runtime vs. naive multi-search.
5. Visualize the failure links in a graph.

Once you grasp Aho-Corasick, you'll see pattern search not as a loop , but as a machine that reads and recognizes.

63. Suffix Structures (Suffix Array, Suffix Tree, LCP)

Suffix-based data structures are among the most powerful tools in string algorithms. They enable fast searching, substring queries, pattern matching, and lexicographic operations , all from one fundamental idea:

> Represent all suffixes of a string in a structured form.

In this section, we explore three key constructs:

- Suffix Array (SA) - lexicographically sorted suffix indices- Longest Common Prefix (LCP) array - shared prefix lengths between neighbors- Suffix Tree - compressed trie of all suffixes Together, they power many advanced algorithms in text processing, bioinformatics, and compression.

1. Suffix Array (SA)

A suffix array stores all suffixes of a string in lexicographic order, represented by their starting indices.

Example: String `banana$` All suffixes:

Index	Suffix
0	banana$
1	anana$
2	nana$
3	ana$
4	na$
5	a$
6	$

Sort them:

Sorted Order	Suffix	Index
0	`$`	6
1	`a$`	5
2	`ana$`	3
3	`anana$`	1
4	`banana$`	0
5	`na$`	4
6	`nana$`	2

Suffix Array: `[6, 5, 3, 1, 0, 4, 2]`

Construction (Prefix Doubling)

We iteratively sort suffixes by first 2 characters, using radix sort on pairs of ranks.

Steps:

1. Assign initial rank by character.
2. Sort by (rank[i], rank[i+k]).
3. Repeat doubling $k \leftarrow 2k$ until all ranks distinct.

Time Complexity: ($On \log n$) Space: (O(n))

Tiny Code (C, Sketch)

```
typedef struct { int idx, rank[2]; } Suffix;
int cmp(Suffix a, Suffix b) {
    return (a.rank[0]==b.rank[0]) ? (a.rank[1]-b.rank[1]) : (a.rank[0]-b.rank[0]);
}

void buildSA(char *s, int n, int sa[]) {
    Suffix suf[n];
    for (int i = 0; i < n; i++) {
        suf[i].idx = i;
        suf[i].rank[0] = s[i];
        suf[i].rank[1] = (i+1<n) ? s[i+1] : -1;
    }
    for (int k = 2; k < 2*n; k *= 2) {
        qsort(suf, n, sizeof(Suffix), cmp);
        int r = 0, rank[n]; rank[suf[0].idx]=0;
        for (int i=1;i<n;i++) {
            if (suf[i].rank[0]!=suf[i-1].rank[0] || suf[i].rank[1]!=suf[i-1].rank[1]) r++;
            rank[suf[i].idx]=r;
        }
        for (int i=0;i<n;i++){
            suf[i].rank[0] = rank[suf[i].idx];
            suf[i].rank[1] = (suf[i].idx+k/2<n)?rank[suf[i].idx+k/2]:-1;
        }
    }
    for (int i=0;i<n;i++) sa[i]=suf[i].idx;
}
```

2. Longest Common Prefix (LCP)

The LCP array stores the length of the longest common prefix between consecutive suffixes in SA order.

Example: `banana$`

SA	Suffix	LCP
6	$	0
5	a$	0
3	ana$	1
1	anana$	3
0	banana$	0
4	na$	0
2	nana$	2

So LCP = `[0,0,1,3,0,0,2]`

Kasai's Algorithm (Build in O(n))

We compute LCP in one pass using inverse SA:

```
void buildLCP(char *s, int n, int sa[], int lcp[]) {
    int rank[n];
    for (int i=0;i<n;i++) rank[sa[i]]=i;
    int k=0;
    for (int i=0;i<n;i++) {
        if (rank[i]==n-1) { k=0; continue; }
        int j = sa[rank[i]+1];
        while (i+k<n && j+k<n && s[i+k]==s[j+k]) k++;
        lcp[rank[i]]=k;
        if (k>0) k--;
    }
}
```

Time Complexity: (O(n))

3. Suffix Tree

A suffix tree is a compressed trie of all suffixes.

Each edge holds a substring interval, not individual characters. This gives:

- Construction in (O(n)) (Ukkonen's algorithm)- Pattern search in (O(m))- Many advanced uses (e.g., longest repeated substring) Example: String: `banana$` Suffix tree edges:

```
(root)
  b[0:0] → ...
  a[1:1] → ...
  n[2:2] → ...
```

Edges compress consecutive letters into intervals like `[start:end]`.

Comparison

Structure	Space	Build Time	Search
Suffix Array	(O(n))	(O$n \log n$)	(O$m \log n$)
LCP Array	(O(n))	(O(n))	Range queries
Suffix Tree	(O(n))	(O(n))	(O(m))

Suffix Array + LCP   compact Suffix Tree.

4. Applications

1. Substring search - binary search in SA
2. Longest repeated substring - max(LCP)
3. Lexicographic order - direct from SA
4. Distinct substrings count = (n(n+1)/2 - LCP[i])
5. Pattern frequency - range query in SA using LCP

5. Tiny Code (Search via SA)

```
int searchSA(char *t, int n, char *p, int sa[]) {
    int l=0, r=n-1, m=strlen(p);
    while (l <= r) {
        int mid = (l+r)/2;
        int cmp = strncmp(t+sa[mid], p, m);
        if (cmp==0) return sa[mid];
        else if (cmp<0) l=mid+1;
        else r=mid-1;
    }
    return -1;
}
```

6. Summary

Concept	Purpose	Complexity
Suffix Array	Sorted suffix indices	($On\log n$)
LCP Array	Adjacent suffix overlap	(O(n))
Suffix Tree	Compressed trie of suffixes	(O(n))

Together they form the core of advanced string algorithms.

Why It Matters

Suffix structures reveal hidden order in strings. They turn raw text into searchable, analyzable data , ideal for compression, search engines, and DNA analysis.

> "All suffixes, perfectly sorted , the DNA of text."

Try It Yourself

1. Build suffix array for `banana$` by hand.
2. Write code to compute LCP and longest repeated substring.
3. Search multiple patterns using binary search on SA.
4. Count distinct substrings from SA + LCP.
5. Compare SA-based vs. tree-based search performance.

Mastering suffix structures equips you to tackle problems that were once "too big" for brute force , now solvable with elegance and order.

64. Palindromes and Periodicity (Manacher)

Palindromes are symmetric strings that read the same forwards and backwards , like “level”, “racecar”, or “madam”. They arise naturally in text analysis, bioinformatics, and even in data compression.

This section introduces efficient algorithms to detect and analyze palindromic structure and periodicity in strings, including the legendary Manacher’s Algorithm, which finds all palindromic substrings in linear time.

1. What Is a Palindrome?

A string (S) is a palindrome if:

$$S[i] = S[n - i + 1] \quad \text{for all } i$$

Examples:

- `"abba"` is even-length palindrome- `"aba"` is odd-length palindrome A string may contain many palindromic substrings , our goal is to find all centers efficiently.

2. Naive Approach

For each center (between characters or at characters), expand outward while characters match.

```
for each center c:
    expand left, right while S[l] == S[r]
```

Complexity: (On^2) , too slow for large strings.

We need something faster , that’s where Manacher’s Algorithm steps in.

3. Manacher’s Algorithm (O(n))

Manacher’s Algorithm finds the radius of the longest palindrome centered at each position in linear time.

It cleverly reuses previous computations using mirror symmetry and a current right boundary.

Step-by-Step

1. Preprocess string to handle even-length palindromes: Insert `#` between characters.

   Example:

   ```
   S = "abba"
   T = "^#a#b#b#a#$"
   ```

   (`^` and `$` are sentinels)

2. Maintain:

   - `C`: center of rightmost palindrome - `R`: right boundary - `P[i]`: palindrome radius at `i`

3. For each position `i`:

   - mirror position `mirror = 2*C - i` - initialize `P[i] = min(R - i, P[mirror])` - expand around `i` while characters match - if new palindrome extends past `R`, update `C` and `R`

4. The **maximum value of `P[i]`** gives the longest palindrome.

Example

```
S = "abba"
T = "^#a#b#b#a#$"
P = [0,0,1,0,3,0,3,0,1,0,0]
Longest radius = 3 → "abba"
```

Tiny Code (C Implementation)

```c
int manacher(char *s) {
    int n = strlen(s);
    char t[2*n + 3];
    int p[2*n + 3];
    int m = 0;
    t[m++] = '^';
    for (int i=0;i<n;i++) {
        t[m++] = '#';
        t[m++] = s[i];
    }
    t[m++] = '#'; t[m++] = '$';
    t[m] = '\0';
```

```
    int c = 0, r = 0, maxLen = 0;
    for (int i=1; i<m-1; i++) {
        int mirror = 2*c - i;
        if (i < r)
            p[i] = (r - i < p[mirror]) ? (r - i) : p[mirror];
        else p[i] = 0;
        while (t[i + 1 + p[i]] == t[i - 1 - p[i]])
            p[i]++;
        if (i + p[i] > r) {
            c = i;
            r = i + p[i];
        }
        if (p[i] > maxLen) maxLen = p[i];
    }
    return maxLen;
}
```

Time Complexity: (O(n)) Space: (O(n))

4. Periodicity and Repetition

A string (S) has a period (p) if:

$$S[i] = S[i + p] \text{ for all valid } i$$

Example: `abcabcabc` has period 3 (`abc`).

Checking Periodicity:

1. Build prefix function (as in KMP).
2. Let (n = |S|), $p = n - \pi[n-1]$.
3. If $n \mod p = 0$, period = (p).

Example:

```
S = "ababab"
  = [0,0,1,2,3,4]
p = 6 - 4 = 2
6 mod 2 = 0 → periodic
```

Tiny Code (Check Periodicity)

```
int period(char *s) {
    int n = strlen(s), pi[n];
    pi[0]=0;
    for(int i=1,k=0;i<n;i++){
        while(k>0 && s[k]!=s[i]) k=pi[k-1];
        if(s[k]==s[i]) k++;
        pi[i]=k;
    }
    int p = n - pi[n-1];
    return (n % p == 0) ? p : n;
}
```

5. Applications

- Palindrome Queries: is substring (S[l:r]) palindrome? → precompute radii- Longest Palindromic Substring- DNA Symmetry Analysis- Pattern Compression / Period Detection- String Regularity Tests

6. Summary

Concept	Purpose	Time
Naive Expand	Simple palindrome check	(On^2)
Manacher	Longest palindromic substring	(O(n))
KMP Prefix	Period detection	(O(n))

Why It Matters

Palindromes reveal hidden symmetries. Manacher's algorithm is a gem , a linear-time mirror-based solution to a quadratic problem.

> "In every word, there may hide a reflection."

Try It Yourself

1. Run Manacher's algorithm on `"abacdfgdcaba"`.
2. Modify code to print all palindromic substrings.
3. Use prefix function to find smallest period.

4. Combine both to find palindromic periodic substrings.
5. Compare runtime vs. naive expand method.

Understanding palindromes and periodicity teaches how structure emerges from repetition , a central theme in all of algorithmic design.

65. Edit Distance and Alignment

Edit distance measures how different two strings are , the minimal number of operations needed to turn one into the other. It's a cornerstone of spell checking, DNA sequence alignment, plagiarism detection, and fuzzy search.

The most common form is the Levenshtein distance, using:

- Insertion (add a character)- Deletion (remove a character)- Substitution (replace a character) We'll also touch on alignment, which generalizes this idea with custom scoring and penalties.

1. Problem Definition

Given two strings (A) and (B), find the minimum number of edits to convert $A \to B$.

If (A = "kitten") (B = "sitting")

One optimal sequence:

```
kitten → sitten (substitute 'k'→'s')
sitten → sittin (substitute 'e'→'i')
sittin → sitting (insert 'g')
```

So edit distance = 3.

2. Dynamic Programming Solution

Let $dp[i][j]$ be the minimum edits to convert $A[0..i-1] \to B[0..j-1]$.

Recurrence:

$$dp[i][j] = \begin{cases} dp[i-1][j-1], & \text{if } A[i-1] = B[j-1], \\ 1 + \min\left(dp[i-1][j],\, dp[i][j-1],\, dp[i-1][j-1]\right), & \text{otherwise} \end{cases}$$

Where: - $dp[i-1][j]$: delete from A - $dp[i][j-1]$: insert into A - $dp[i-1][j-1]$: substitute

Base cases:

$$dp[0][j] = j, \quad dp[i][0] = i$$

Time complexity: $O(|A||B|)$

Example

A = `kitten`, B = `sitting`

	“ ”	s	i	t	t	i	n	g
“ ”	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	3

Edit distance = 3

Tiny Code (C)

```c
#include <stdio.h>
#include <string.h>
#define MIN3(a,b,c) ((a<b)?((a<c)?a:c):((b<c)?b:c))

int edit_distance(char *A, char *B) {
    int n = strlen(A), m = strlen(B);
    int dp[n+1][m+1];
    for (int i=0;i<=n;i++) dp[i][0]=i;
    for (int j=0;j<=m;j++) dp[0][j]=j;
    for (int i=1;i<=n;i++)
        for (int j=1;j<=m;j++)
            if (A[i-1]==B[j-1])
                dp[i][j]=dp[i-1][j-1];
            else
                dp[i][j]=1+MIN3(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]);
    return dp[n][m];
}

int main() {
    printf("%d\n", edit_distance("kitten","sitting")); // 3
}
```

3. Space Optimization

We only need the previous row to compute the current row.

So,
Space complexity: $O(\min(|A|,|B|))$

```
int edit_distance_opt(char *A, char *B) {
    int n=strlen(A), m=strlen(B);
    int prev[m+1], curr[m+1];
    for(int j=0;j<=m;j++) prev[j]=j;
    for(int i=1;i<=n;i++){
        curr[0]=i;
        for(int j=1;j<=m;j++){
            if(A[i-1]==B[j-1]) curr[j]=prev[j-1];
            else curr[j]=1+MIN3(prev[j], curr[j-1], prev[j-1]);
        }
        memcpy(prev,curr,sizeof(curr));
    }
    return prev[m];
}
```

4. Alignment

Alignment shows which characters correspond between two strings. Used in bioinformatics (e.g., DNA sequence alignment).

Each operation has a cost:

- Match: 0
- Mismatch: 1
- Gap (insert/delete): 1 We fill the DP table similarly, but track choices to trace back alignment.

Example Alignment

```
A: kitten-
B: sitt-ing
```

We can visualize the transformation path by backtracking dp table.

Scoring Alignment (General Form)

We can generalize:

$$dp[i][j] = \min \Big\{ dp[i-1][j-1] + cost(A_i, B_j)\ dp[i-1][j] + gap\ dp[i][j-1] + gap$$

Used in Needleman-Wunsch (global alignment) and Smith-Waterman (local alignment).

5. Variants

- Damerau-Levenshtein: adds transposition (swap adjacent chars)- Hamming Distance: only substitutions, equal-length strings- Weighted Distance: different operation costs- Local Alignment: only best matching substrings

6. Summary

Method	Operations	Time	Use
Levenshtein	insert, delete, replace	(O(nm))	Spell check, fuzzy search
Hamming	substitution only	(O(n))	DNA, binary strings
Alignment (Needleman-Wunsch)	with scoring	(O(nm))	Bioinformatics
Local Alignment (Smith-Waterman)	best substring	(O(nm))	DNA regions

Why It Matters

Edit distance transforms "difference" into data. It quantifies how far apart two strings are, enabling flexible, robust comparisons.

> "Similarity isn't perfection , it's the cost of becoming alike."

Try It Yourself

1. Compute edit distance between "intention" and "execution".
2. Trace back operations to show alignment.
3. Modify costs (insertion=2, deletion=1, substitution=2) and compare results.
4. Implement Hamming distance for equal-length strings.
5. Explore Smith-Waterman for longest common substring.

Once you master edit distance, you can build tools that understand typos, align genomes, and search imperfectly , perfectly.

66. Compression (Huffman, Arithmetic, LZ77, BWT)

Compression algorithms let us encode information efficiently, reducing storage or transmission cost without losing meaning. They turn patterns and redundancy into shorter representations , the essence of data compression.

This section introduces the key families of lossless compression algorithms that form the backbone of formats like ZIP, PNG, and GZIP.

We'll explore:

- Huffman Coding (prefix-free variable-length codes)
- Arithmetic Coding (fractional interval encoding)
- LZ77 / LZ78 (dictionary-based methods)
- Burrows-Wheeler Transform (BWT) (reversible sorting transform)

1. Huffman Coding

Huffman coding assigns shorter codes to frequent symbols, and longer codes to rare ones , achieving optimal compression among prefix-free codes.

A. Algorithm

1. Count frequencies of all symbols.
2. Build a min-heap of nodes `(symbol, freq)`.
3. While heap size > 1:
    - Extract two smallest nodes `a`, `b`. - Create new node with `freq = a.freq + b.freq`. - Push back into heap.4. Assign `0` to left, `1` to right.
4. Traverse tree , collect codes.

Each symbol gets a unique prefix code (no code is prefix of another).

B. Example

Text: `ABRACADABRA`

Frequencies:

Symbol	Count
A	5
B	2
R	2
C	1
D	1

Building tree gives codes like:

```
A: 0
B: 101
R: 100
C: 1110
D: 1111
```

Encoded text: `0 101 100 0 1110 0 1111 0 101 100 0` Compression achieved!

Tiny Code (C, Sketch)

```c
typedef struct Node {
    char ch;
    int freq;
    struct Node *left, *right;
} Node;
```

Use a min-heap (priority queue) to build the tree. Traverse recursively to print codewords.

Complexity: $(On \log n)$

2. Arithmetic Coding

Instead of mapping symbols to bit strings, arithmetic coding maps the entire message to a single number in [0,1).

We start with interval ([0,1)), and iteratively narrow it based on symbol probabilities.

Example

Symbols: {A: 0.5, B: 0.3, C: 0.2} Message: `ABC`

Intervals:

```
Start: [0, 1)
A → [0, 0.5)
B → [0.25, 0.4)
C → [0.34, 0.37)
```

Final code = any number in [0.34, 0.37) (e.g. 0.35)

Decoding reverses this process.

Advantage: achieves near-optimal entropy compression. Used in: JPEG2000, H.264

Time Complexity: (O(n))

3. LZ77 (Sliding Window Compression)

LZ77 replaces repeated substrings with back-references `(offset, length, next_char)` pointing into a sliding window.

Example

Text: `abcabcabcx`

Window slides; when `abc` repeats:

```
(0,0,'a'), (0,0,'b'), (0,0,'c'),
(3,3,'x')  // "abc" repeats from 3 chars back
```

So sequence is compressed as references to earlier substrings.

Used in: DEFLATE (ZIP, GZIP), PNG

Time: (O(n)), Space: proportional to window size.

Tiny Code (Conceptual)

```
struct Token { int offset, length; char next; };
```

Search previous window for longest match before emitting token.

4. LZ78 (Dictionary-Based)

Instead of sliding window, LZ78 builds an explicit dictionary of substrings.

Algorithm:

- Start with empty dictionary.- Read input, find longest prefix in dictionary.- Output `(index, next_char)` and insert new entry. Example:

```
Input: ABAABABAABAB
Output: (0,A), (0,B), (1,B), (2,A), (4,A), (3,B)
```

Used in: LZW (GIF, TIFF)

5. Burrows-Wheeler Transform (BWT)

BWT is not compression itself , it permutes text to cluster similar characters, making it more compressible by run-length or Huffman coding.

Steps

1. Generate all rotations of string.
2. Sort them lexicographically.
3. Take last column as output.

Example: `banana$`

Rotations	Sorted
banana$	$banana\|\|anana$b
a$banan\|na$bana	
$banana\|nana$ba	

Last column: `annb$aa` BWT(“banana") = "$annb$aa”

Reversible with index of original row.

Used in: bzip2, FM-index (bioinformatics)

6. Summary

Algorithm	Idea	Complexity	Use
Huffman	Variable-length prefix codes	(O$n \log n$)	General compression
Arithmetic	Interval encoding	(O(n))	Near-optimal entropy
LZ77	Sliding window matches	(O(n))	ZIP, PNG
LZ78	Dictionary building	(O(n))	GIF, TIFF
BWT	Permute for clustering	(O$n \log n$)	bzip2

Why It Matters

Compression algorithms reveal structure in data , they exploit patterns that humans can't see. They're also a window into information theory, showing how close we can get to the entropy limit.

> "To compress is to understand , every bit saved is a pattern found."

Try It Yourself

1. Build a Huffman tree for `MISSISSIPPI`.
2. Implement a simple LZ77 encoder for repeating patterns.
3. Apply BWT and observe clustering of symbols.
4. Compare Huffman and Arithmetic outputs on same input.
5. Explore DEFLATE format combining LZ77 + Huffman.

Understanding compression means learning to see redundancy , the key to efficient storage, transmission, and understanding itself.

67. Cryptographic Hashes and Checksums

In algorithms, hashing helps us map data to fixed-size values. But when used for security and verification, hashing becomes a cryptographic tool. This section explores cryptographic hashes and checksums , algorithms that verify integrity, detect corruption, and secure data.

We'll look at:

- Simple checksums (parity, CRC)- Cryptographic hash functions (MD5, SHA family, BLAKE3)- Properties like collision resistance and preimage resistance- Practical uses in verification, signing, and storage

1. Checksums

Checksums are lightweight methods to detect accidental errors in data (not secure against attackers). They're used in filesystems, networking, and storage to verify integrity.

A. Parity Bit

Adds one bit to make total 1s even or odd. Used in memory or communication to detect single-bit errors.

Example: Data = `1011` → has three 1s. Add parity bit `1` to make total 4 (even parity).

Limitation: Only detects odd number of bit errors.

B. Modular Sum (Simple Checksum)

Sum all bytes (mod 256 or 65536).

Tiny Code (C)

```c
uint8_t checksum(uint8_t *data, int n) {
    uint32_t sum = 0;
    for (int i = 0; i < n; i++) sum += data[i];
    return (uint8_t)(sum % 256);
}
```

Use: Simple file or packet validation.

C. CRC (Cyclic Redundancy Check)

CRCs treat bits as coefficients of a polynomial. Divide by a generator polynomial, remainder = CRC code.

Used in Ethernet, ZIP, and PNG.

Example: CRC-32, CRC-16.

Fast hardware and table-driven implementations available.

Key Property:

- Detects most burst errors- Not cryptographically secure

2. Cryptographic Hash Functions

A cryptographic hash function (h(x)) maps any input to a fixed-size output such that:

1. Deterministic: same input → same output
2. Fast computation
3. Preimage resistance: hard to find (x) given (h(x))
4. Second-preimage resistance: hard to find $x' \neq x$ with (h(x') = h(x))
5. Collision resistance: hard to find any two distinct inputs with same hash

Algorithm	Output (bits)	Notes
MD5	128	Broken (collisions found)
SHA-1	160	Deprecated
SHA-256	256	Standard (SHA-2 family)
SHA-3	256	Keccak-based sponge
BLAKE3	256	Fast, parallel, modern

Example

```
h("hello") = 2cf24dba5fb0a... (SHA-256)
```

Change one letter, hash changes completely (avalanche effect):

```
h("Hello") = 185f8db32271f...
```

Even small changes → big differences.

Tiny Code (C, using pseudo-interface)

```c
#include <openssl/sha.h>

unsigned char hash[SHA256_DIGEST_LENGTH];
SHA256((unsigned char*)"hello", 5, hash);
```

Print hash as hex string to verify.

3. Applications

- Data integrity: verify files (e.g., SHA256SUM)- Digital signatures: sign hashes, not raw data- Password storage: store hashes, not plaintext- Deduplication: detect identical files via hashes- Blockchain: link blocks with hash pointers- Git: stores objects via SHA-1 identifiers

4. Hash Collisions

A collision occurs when (h(x) = h(y)) for $x \neq y$. Good cryptographic hashes make this computationally infeasible.

By the birthday paradox, collisions appear after $2^{n/2}$ operations for an (n)-bit hash.

Hence, SHA-256 → ~2^{128} effort to collide.

5. Checksums vs Hashes

Feature	Checksum	Cryptographic Hash
Goal	Detect errors	Ensure integrity and authenticity
Resistance	Low	High
Output Size	Small	128-512 bits
Performance	Very fast	Fast but secure
Example	CRC32	SHA-256, BLAKE3

Why It Matters

Checksums catch accidental corruption, hashes protect against malicious tampering. Together, they guard the trustworthiness of data , the foundation of secure systems.

> "Integrity is invisible , until it's lost."

Try It Yourself

1. Compute CRC32 of a text file, flip one bit, and recompute.
2. Use `sha256sum` to verify file integrity.
3. Experiment: change one character in input, observe avalanche.
4. Compare performance of SHA-256 and BLAKE3.
5. Research how Git uses SHA-1 to track versions.

By learning hashes, you master one of the pillars of security , proof that something hasn't changed, even when everything else does.

68. Approximate and Streaming Matching

Exact string matching (like KMP or Boyer-Moore) demands perfect alignment between pattern and text. But what if errors, noise, or incomplete data exist?

That's where approximate matching and streaming matching come in. These algorithms let you search efficiently even when:

- The pattern might contain typos or mutations- The text arrives in a stream (too large to store entirely)- You want to match "close enough," not "exactly" They're crucial in search engines, spell checkers, bioinformatics, and real-time monitoring systems.

1. Approximate String Matching

Approximate string matching finds occurrences of a pattern in a text allowing mismatches, insertions, or deletions , often measured by edit distance.

A. Dynamic Programming (Levenshtein Distance)

Given two strings A and B, the edit distance is the minimum number of insertions, deletions, or substitutions to turn A into B.

We can build a DP table $dp[i][j]$:

- $dp[i][0] = i$ (delete all characters)
- $dp[0][j] = j$ (insert all characters)
- If $A[i] = B[j]$, then $dp[i][j] = dp[i-1][j-1]$
- Else $dp[i][j] = 1 + \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])$

Tiny Code (C)

```c
int edit_distance(char *a, char *b) {
    int n = strlen(a), m = strlen(b);
    int dp[n+1][m+1];
    for (int i = 0; i <= n; i++) dp[i][0] = i;
    for (int j = 0; j <= m; j++) dp[0][j] = j;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (a[i-1] == b[j-1]) dp[i][j] = dp[i-1][j-1];
            else dp[i][j] = 1 + fmin(fmin(dp[i-1][j], dp[i][j-1]), dp[i-1][j-1]);
    return dp[n][m];
}
```

This computes Levenshtein distance in (O(nm)) time.

B. Bitap Algorithm (Shift-Or)

When pattern length is small, Bitap uses bitmasks to track mismatches. It efficiently supports up to k errors and runs in near linear time for small patterns.

Used in grep -E, ag, and fuzzy searching systems.

Idea: Maintain a bitmask where 1 = mismatch, 0 = match. Shift and OR masks as we scan text.

C. k-Approximate Matching

Find all positions where edit distance k. Efficient for small (k) (e.g., spell correction: edit distance 2).

Applications:

- Typo-tolerant search- DNA sequence matching- Autocomplete systems

2. Streaming Matching

In streaming, the text is too large or unbounded, so we must process input online. We can't store everything , only summaries or sketches.

A. Rolling Hash (Rabin-Karp style)

Maintains a moving hash of recent characters. When new character arrives, update hash in (O(1)). Compare with pattern's hash for possible match.

Good for sliding window matching.

Example:

```
hash = (base * (hash - old_char * base^(m-1)) + new_char) % mod;
```

B. Fingerprinting (Karp-Rabin Fingerprint)

A compact representation of a substring. If fingerprints match, do full verification (avoid false positives). Used in streaming algorithms and chunking.

C. Sketch-Based Matching

Algorithms like Count-Min Sketch or SimHash build summaries of large data. They help approximate similarity between streams.

Applications:

- Near-duplicate detection (SimHash in Google)- Network anomaly detection- Real-time log matching

3. Approximate Matching in Practice

Domain	Use Case	Algorithm
Spell Checking	"recieve" → "receive"	Edit Distance
DNA Alignment	Find similar sequences	Smith-Waterman
Autocomplete	Suggest close matches	Fuzzy Search
Logs & Streams	Online pattern alerts	Streaming Bitap, Karp-Rabin
Near-Duplicate	Detect similar text	SimHash, MinHash

Algorithm	Time	Space	Notes

4. Complexity

Algorithm	Time	Space	Notes
Levenshtein DP	(O(nm))	(O(nm))	Exact distance
Bitap	(O(n))	(O(1))	For small patterns
Rolling Hash	(O(n))	(O(1))	Probabilistic match
SimHash	(O(n))	(O(1))	Approximate similarity

Why It Matters

Real-world data is messy , typos, noise, loss, corruption. Approximate matching lets you build algorithms that forgive errors and adapt to streams. It powers everything from search engines to genomics, ensuring your algorithms stay practical in an imperfect world.

Try It Yourself

1. Compute edit distance between "kitten" and "sitting."
2. Implement fuzzy search that returns words with 1 typo.
3. Use rolling hash to detect repeated substrings in a stream.
4. Experiment with SimHash to compare document similarity.
5. Observe how small typos affect fuzzy vs exact search.

69. Bioinformatics Alignment (Needleman-Wunsch, Smith-Waterman)

In bioinformatics, comparing DNA, RNA, or protein sequences is like comparing strings , but with biological meaning. Each sequence is made of letters (A, C, G, T for DNA; amino acids for proteins). To analyze similarity, scientists use sequence alignment algorithms that handle insertions, deletions, and substitutions.

Two fundamental methods dominate:

- Needleman-Wunsch for global alignment- Smith-Waterman for local alignment

1. Sequence Alignment

Alignment means placing two sequences side by side to maximize matches and minimize gaps or mismatches.

For example:

```
A C G T G A
| | |   | |
A C G A G A
```

Here, mismatches and gaps may occur, but the alignment finds the best possible match under a scoring system.

Scoring System

Alignment uses scores instead of just counts. Typical scheme:

- Match: +1- Mismatch: -1- Gap (insertion or deletion): -2 You can adjust weights depending on the biological context.

2. Needleman-Wunsch (Global Alignment)

Used when you want to align entire sequences , from start to end.

It uses dynamic programming to build a score table (dp[i][j]), where each cell represents the best score for aligning prefixes (A[1..i]) and (B[1..j]).

Recurrence:

$$dp[i][j] = \max \left\{ dp[i-1][j-1] + \text{score}(A_i, B_j)\ dp[i-1][j] + \text{gap penalty}\ dp[i][j-1] + \text{gap penalty} \right.$$

Base cases:

$$dp[0][j] = j \times \text{gap penalty}, \quad dp[i][0] = i \times \text{gap penalty}$$

Tiny Code (C)

```c
int max3(int a, int b, int c) {
    return a > b ? (a > c ? a : c) : (b > c ? b : c);
}

int needleman_wunsch(char *A, char *B, int match, int mismatch, int gap) {
    int n = strlen(A), m = strlen(B);
    int dp[n+1][m+1];
    for (int i = 0; i <= n; i++) dp[i][0] = i * gap;
    for (int j = 0; j <= m; j++) dp[0][j] = j * gap;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            int s = (A[i-1] == B[j-1]) ? match : mismatch;
            dp[i][j] = max3(dp[i-1][j-1] + s, dp[i-1][j] + gap, dp[i][j-1] + gap);
        }
    }
    return dp[n][m];
}
```

Example:

```
A = "ACGT"
B = "AGT"
match = +1, mismatch = -1, gap = -2
```

Produces optimal alignment:

```
A C G T
A - G T
```

3. Smith-Waterman (Local Alignment)

Used when sequences may have similar segments, not full-length similarity. Perfect for finding motifs or conserved regions.

Recurrence is similar, but with local reset to zero:

$$dp[i][j] = \max \left\{0,\ dp[i-1][j-1] + \text{score}(A_i, B_j),\ dp[i-1][j] + \text{gap penalty},\ dp[i][j-1] + \text{gap penalty}\right.$$

Final answer = maximum value in the table (not necessarily at the end).

It finds the best substring alignment.

Example

```
A = "ACGTTG"
B = "CGT"
```

Smith-Waterman finds best local match:

```
A C G T
  | | |
  C G T
```

Unlike global alignment, extra prefixes or suffixes are ignored.

4. Variants and Extensions

Algorithm	Type	Notes
Needleman-Wunsch	Global	Aligns full sequences
Smith-Waterman	Local	Finds similar subsequences
Gotoh Algorithm	Global	Uses affine gap penalty (opening + extension)
BLAST	Heuristic	Speeds up search for large databases

BLAST (Basic Local Alignment Search Tool) uses word seeds and extension, trading exactness for speed , essential for large genome databases.

5. Complexity

Both Needleman-Wunsch and Smith-Waterman run in:

- Time: (O(nm))- Space: (O(nm)) But optimized versions use banded DP or Hirschberg's algorithm to cut memory to (O(n + m)).

Why It Matters

Sequence alignment bridges computer science and biology. It's how we:

- Compare species- Identify genes- Detect mutations- Trace ancestry- Build phylogenetic trees The idea of "minimum edit cost" echoes everywhere , from spell checkers to DNA analysis.

  "In biology, similarity is a story. Alignment is how we read it."

Try It Yourself

1. Implement Needleman-Wunsch for short DNA sequences.
2. Change gap penalties , see how alignment shifts.
3. Compare outputs from global and local alignment.
4. Use real sequences from GenBank to test.
5. Explore BLAST online and compare to exact alignment results.

70. Text Indexing and Search Structures

When text becomes large , think books, databases, or the entire web , searching naively for patterns (O(nm)) is far too slow. We need indexing structures that let us search fast, often in O(m) or O(log n) time.

This section covers the backbone of search engines and string processing:

- Suffix Arrays- Suffix Trees- Inverted Indexes- Tries and Prefix Trees- Compressed Indexes like FM-Index (Burrows-Wheeler)

1. Why Index?

A text index is like a table of contents , it doesn't store the book, but lets you jump straight to what you want.

If you have a text of length (n), and you'll run many queries, it's worth building an index (even if it costs (O$n \log n$) to build).

Without indexing: each query takes (O(nm)). With indexing: each query can take (O(m)) or less.

2. Suffix Array

A suffix array is a sorted array of all suffixes of a string.

For text `"banana"`, suffixes are:

```
0: banana
1: anana
2: nana
3: ana
4: na
5: a
```

Sorted lexicographically:

```
5: a
3: ana
1: anana
0: banana
4: na
2: nana
```

Suffix Array = `[5, 3, 1, 0, 4, 2]`

To search, binary search over the suffix array using your pattern , ($Om \log n$).

Tiny Code (C) (naive construction)

```c
int cmp(const void *a, const void *b, void *txt) {
    int i = *(int*)a, j = *(int*)b;
    return strcmp((char*)txt + i, (char*)txt + j);
}

void build_suffix_array(char *txt, int n, int sa[]) {
    for (int i = 0; i < n; i++) sa[i] = i;
    qsort_r(sa, n, sizeof(int), cmp, txt);
}
```

Modern methods like prefix doubling or radix sort build it in ($On \log n$).

Applications:

- Fast substring search- Longest common prefix (LCP) array- Pattern matching in DNA sequences- Plagiarism detection

3. Suffix Tree

A suffix tree is a compressed trie of all suffixes , each edge stores multiple characters.

For `"banana"`, you'd build a tree where each leaf corresponds to a suffix index.

Advantages:

- Pattern search in (O(m))- Space (O(n)) (with compression) Built using Ukkonen's algorithm in (O(n)).

Use Suffix Array + LCP as a space-efficient alternative.

4. FM-Index (Burrows-Wheeler Transform)

Used in compressed full-text search (e.g., Bowtie, BWA). Combines:

- Burrows-Wheeler Transform (BWT)- Rank/select bitvectors Supports pattern search in O(m) time with very low memory.

Idea: transform text so similar substrings cluster together, enabling compression and backward search.

Applications:

- DNA alignment- Large text archives- Memory-constrained search

5. Inverted Index

Used in search engines. Instead of suffixes, it indexes words.

For example, text corpus:

```
doc1: quick brown fox
doc2: quick red fox
```

Inverted index:

```
"quick" → [doc1, doc2]
"brown" → [doc1]
"red"   → [doc2]
"fox"   → [doc1, doc2]
```

Now searching “quick fox” becomes set intersection of lists.

Used with ranking functions (TF-IDF, BM25).

6. Tries and Prefix Trees

A trie stores strings character by character. Each node = prefix.

```c
typedef struct Node {
    struct Node *child[26];
    int end;
} Node;
```

Perfect for:

- Autocomplete- Prefix search- Spell checkers Search: O(m), where m = pattern length.

Compressed tries (Patricia trees) reduce space.

7. Comparing Structures

Structure	Search Time	Build Time	Space	Notes
Trie	O(m)	O(n)	High	Prefix queries
Suffix Array	O(m log n)	O(n log n)	Medium	Sorted suffixes
Suffix Tree	O(m)	O(n)	High	Rich structure
FM-Index	O(m)	O(n)	Low	Compressed
Inverted Index	O(k)	O(N)	Medium	Word-based

Why It Matters

Text indexing is the backbone of search engines, DNA alignment, and autocomplete systems. Without it, Google searches, code lookups, or genome scans would take minutes, not milliseconds.

> "Indexing turns oceans of text into navigable maps."

Try It Yourself

1. Build a suffix array for "banana" and perform binary search for "ana."
2. Construct a trie for a dictionary and query prefixes.
3. Write a simple inverted index for a few documents.
4. Compare memory usage of suffix tree vs suffix array.
5. Experiment with FM-index using an online demo (like BWT explorer).

Chapter 8. Geometry, Graphics, and Spatial Algorithms

71. Convex Hull (Graham, Andrew, Chan)

In computational geometry, the convex hull of a set of points is the smallest convex polygon that contains all the points. Intuitively, imagine stretching a rubber band around a set of nails on a board , the shape the band takes is the convex hull.

Convex hulls are foundational for many geometric algorithms, like closest pair, Voronoi diagrams, and collision detection.

In this section, we'll explore three classical algorithms:

- Graham Scan - elegant and simple (O(n log n))- Andrew's Monotone Chain - robust and practical (O(n log n))- Chan's Algorithm - advanced and optimal (O(n log h), where h = number of hull points)

1. Definition

Given a set of points $P = p_1, p_2, ..., p_n$, the convex hull, (CH(P)), is the smallest convex polygon enclosing all points.

Formally:

$$\text{CH}(P) = \bigcap C \subseteq \mathbb{R}^2 \mid C \text{ is convex and } P \subseteq C$$

A polygon is convex if every line segment between two points of the polygon lies entirely inside it.

2. Orientation Test

All convex hull algorithms rely on an orientation test using cross product: Given three points (a, b, c):

$$\text{cross}(a, b, c) = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

- `> 0` → counter-clockwise turn- `< 0` → clockwise turn- `= 0` → collinear

3. Graham Scan

One of the earliest convex hull algorithms.

Idea:

1. Pick the lowest point (and leftmost if tie).
2. Sort all other points by polar angle with respect to it.
3. Traverse points and maintain a stack:

    - Add point - While last three points make a right turn, pop middle one4. Remaining points form convex hull in CCW order.

Tiny Code (C)

```
typedef struct { double x, y; } Point;

double cross(Point a, Point b, Point c) {
    return (b.x - a.x)*(c.y - a.y) - (b.y - a.y)*(c.x - a.x);
}

int cmp(const void *p1, const void *p2) {
    Point *a = (Point*)p1, *b = (Point*)p2;
    // Compare by polar angle or distance
    return (a->y != b->y) ? (a->y - b->y) : (a->x - b->x);
}

int graham_scan(Point pts[], int n, Point hull[]) {
    qsort(pts, n, sizeof(Point), cmp);
    int top = 0;
    for (int i = 0; i < n; i++) {
        while (top >= 2 && cross(hull[top-2], hull[top-1], pts[i]) <= 0)
            top--;
        hull[top++] = pts[i];
    }
    return top; // number of hull points
}
```

Complexity:

- Sorting: ($On \log n$)- Scanning: (O(n)) → Total: O(n log n)

Example

Input:

```
(0, 0), (1, 1), (2, 2), (2, 0), (0, 2)
```

Hull (CCW):

```
(0,0) → (2,0) → (2,2) → (0,2)
```

4. Andrew's Monotone Chain

Simpler and more robust for floating-point coordinates. Builds lower and upper hulls separately.

Steps:

1. Sort points lexicographically (x, then y).
2. Build lower hull (left-to-right)
3. Build upper hull (right-to-left)
4. Concatenate (excluding duplicates)

Tiny Code (C)

```c
int monotone_chain(Point pts[], int n, Point hull[]) {
    qsort(pts, n, sizeof(Point), cmp);
    int k = 0;
    // Lower hull
    for (int i = 0; i < n; i++) {
        while (k >= 2 && cross(hull[k-2], hull[k-1], pts[i]) <= 0) k--;
        hull[k++] = pts[i];
    }
    // Upper hull
    for (int i = n-2, t = k+1; i >= 0; i--) {
        while (k >= t && cross(hull[k-2], hull[k-1], pts[i]) <= 0) k--;
        hull[k++] = pts[i];
    }
    return k-1; // last point == first point
}
```

Time Complexity: ($On \log n$)

5. Chan's Algorithm

When $h \ll n$, Chan's method achieves (O$n \log h$):

1. Partition points into groups of size (m).
2. Compute hulls for each group (Graham).
3. Merge hulls with Jarvis March (gift wrapping).
4. Choose (m) cleverly ($m = 2^k$) to ensure (O$n \log h$).

Used in: large-scale geometric processing.

6. Applications

Domain	Use
Computer Graphics	Shape boundary, hitboxes
GIS / Mapping	Region boundaries
Robotics	Obstacle envelopes
Clustering	Outlier detection
Data Analysis	Minimal bounding shape

7. Complexity Summary

Algorithm	Time	Space	Notes
Graham Scan	(O$n \log n$)	(O(n))	Simple, classic
Monotone Chain	(O$n \log n$)	(O(n))	Stable, robust
Chan's Algorithm	(O$n \log h$)	(O(n))	Best asymptotic

Why It Matters

Convex hulls are one of the cornerstones of computational geometry. They teach sorting, cross products, and geometric reasoning , and form the basis for many spatial algorithms.

"Every scattered set hides a simple shape. The convex hull is that hidden simplicity."

Try It Yourself

1. Implement Graham Scan for 10 random points.
2. Plot the points and verify the hull.
3. Compare results with Andrew's Monotone Chain.
4. Test with collinear and duplicate points.
5. Explore 3D convex hulls (QuickHull, Gift Wrapping) next.

72. Closest Pair and Segment Intersection

Geometric problems often ask: *what's the shortest distance between two points?* or *do these segments cross?* These are classic building blocks in computational geometry , essential for collision detection, graphics, clustering, and path planning.

This section covers two foundational problems:

- Closest Pair of Points - find two points with minimum Euclidean distance- Segment Intersection - determine if (and where) two line segments intersect

1. Closest Pair of Points

Given (n) points in 2D, find a pair with the smallest distance. The brute force solution is (On^2), but using Divide and Conquer, we can solve it in O(n log n).

A. Divide and Conquer Algorithm

Idea:

1. Sort points by x-coordinate.
2. Split into left and right halves.
3. Recursively find closest pairs in each half (distance = (d)).
4. Merge step: check pairs across the split line within (d).

In merge step, we only need to check at most 6 neighbors per point (by geometric packing).

Tiny Code (C, Sketch)

```c
#include <math.h>
typedef struct { double x, y; } Point;

double dist(Point a, Point b) {
    double dx = a.x - b.x, dy = a.y - b.y;
    return sqrt(dx*dx + dy*dy);
}

double brute_force(Point pts[], int n) {
    double d = 1e9;
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            d = fmin(d, dist(pts[i], pts[j]));
    return d;
}
```

Recursive divide and merge:

```
double closest_pair(Point pts[], int n) {
    if (n <= 3) return brute_force(pts, n);
    int mid = n / 2;
    double d = fmin(closest_pair(pts, mid),
                    closest_pair(pts + mid, n - mid));
    // merge step: check strip points within distance d
    // sort by y, check neighbors
    return d;
}
```

Time Complexity: ($On\log n$)

Example:

Points:

```
(2,3), (12,30), (40,50), (5,1), (12,10), (3,4)
```

Closest pair: (2,3) and (3,4), distance = $\sqrt{2}$

B. Sweep Line Variant

Another method uses a line sweep and a balanced tree to keep active points. As you move from left to right, maintain a window of recent points within (d).

Used in large-scale spatial systems.

Applications

Domain	Use
Clustering	Find nearest neighbors
Robotics	Avoid collisions
GIS	Nearest city search
Networking	Sensor proximity

2. Segment Intersection

Given two segments (AB) and (CD), determine whether they intersect. It's the core of geometry engines and vector graphics systems.

A. Orientation Test

We use the cross product (orientation) test again. Two segments (AB) and (CD) intersect if and only if:

1. The segments straddle each other:

$$\text{orient}(A, B, C) \neq \text{orient}(A, B, D)$$

$$\text{orient}(C, D, A) \neq \text{orient}(C, D, B)$$

2. Special cases for collinear points (check bounding boxes).

Tiny Code (C)

```c
double cross(Point a, Point b, Point c) {
    return (b.x - a.x)*(c.y - a.y) - (b.y - a.y)*(c.x - a.x);
}

int on_segment(Point a, Point b, Point c) {
    return fmin(a.x, b.x) <= c.x && c.x <= fmax(a.x, b.x) &&
           fmin(a.y, b.y) <= c.y && c.y <= fmax(a.y, b.y);
}

int intersect(Point a, Point b, Point c, Point d) {
    double o1 = cross(a, b, c);
    double o2 = cross(a, b, d);
    double o3 = cross(c, d, a);
    double o4 = cross(c, d, b);
    if (o1*o2 < 0 && o3*o4 < 0) return 1; // general case
    if (o1 == 0 && on_segment(a,b,c)) return 1;
    if (o2 == 0 && on_segment(a,b,d)) return 1;
    if (o3 == 0 && on_segment(c,d,a)) return 1;
    if (o4 == 0 && on_segment(c,d,b)) return 1;
    return 0;
}
```

B. Line Sweep Algorithm (Bentley-Ottmann)

For multiple segments, check all intersections efficiently. Algorithm:

1. Sort all endpoints by x-coordinate.
2. Sweep from left to right.
3. Maintain active set (balanced BST).
4. Check neighboring segments for intersections.

Time complexity: $O((n + k)\log n)$, where k is the number of intersections.

Used in CAD, map rendering, and collision systems.

3. Complexity Summary

Problem	Naive	Optimal	Technique
Closest Pair	$O(n^2)$	$O(n \log n)$	Divide & Conquer
Segment Intersection	$O(n^2)$	$O((n + k)\log n)$	Sweep Line

Why It Matters

Geometric algorithms like these teach how to reason spatially , blending math, sorting, and logic. They power real-world systems where precision matters: from self-driving cars to game engines.

> "Every point has a neighbor; every path may cross another , geometry finds the truth in space."

Try It Yourself

1. Implement the closest pair algorithm using divide and conquer.
2. Visualize all pairwise distances , see which pairs are minimal.
3. Test segment intersection on random pairs.
4. Modify for 3D line segments using vector cross products.
5. Try building a line sweep visualizer to catch intersections step-by-step.

73. Line Sweep and Plane Sweep Algorithms

The sweep line (or plane sweep) technique is one of the most powerful paradigms in computational geometry. It transforms complex spatial problems into manageable one-dimensional events , by sweeping a line (or plane) across the input and maintaining a dynamic set of active elements.

This method underlies many geometric algorithms:

- Event sorting → handle things in order- Active set maintenance → track current structure- Updates and queries → respond as the sweep progresses Used for intersection detection, closest pair, rectangle union, computational geometry in graphics and GIS.

1. The Core Idea

Imagine a vertical line sweeping from left to right across the plane. At each "event" (like a point or segment endpoint), we update the set of objects the line currently touches , the active set.

Each event may trigger queries, insertions, or removals.

This approach works because geometry problems often depend only on local relationships between nearby elements as the sweep advances.

A. Sweep Line Template

A general structure looks like this:

```
struct Event { double x; int type; Object *obj; };
sort(events.begin(), events.end());

ActiveSet S;

for (Event e : events) {
    if (e.type == START) S.insert(e.obj);
    else if (e.type == END) S.erase(e.obj);
    else if (e.type == QUERY) handle_query(S, e.obj);
}
```

Sorting ensures events are processed in order of increasing x (or another dimension).

2. Classic Applications

Let's explore three foundational problems solvable by sweep techniques.

A. Segment Intersection (Bentley-Ottmann)

Goal: detect all intersections among (n) line segments.

Steps:

1. Sort endpoints by x-coordinate.
2. Sweep from left to right.
3. Maintain an ordered set of active segments (sorted by y).
4. When a new segment starts, check intersection with neighbors above and below.
5. When segments intersect, record intersection and insert a new event at the x-coordinate of intersection.

Complexity: $O((n + k)\log n)$, where k is the number of intersections.

B. Closest Pair of Points

Sweep line version sorts by x, then slides a vertical line while maintaining active points within a strip of width (d) (current minimum). Only need to check at most 6-8 nearby points in strip.

Complexity: ($On \log n$)

C. Rectangle Union Area

Given axis-aligned rectangles, compute total area covered.

Idea:

- Treat vertical edges as events (entering/exiting rectangles).- Sweep line moves along x-axis.- Maintain y-intervals in active set (using a segment tree or interval tree).- At each step, multiply current width × height of union of active intervals. Complexity: ($On \log n$)

Tiny Code Sketch (C)

```
typedef struct { double x, y1, y2; int type; } Event;
Event events[MAX];
int n_events;

qsort(events, n_events, sizeof(Event), cmp_by_x);

double prev_x = events[0].x, area = 0;
SegmentTree T;

for (int i = 0; i < n_events; i++) {
    double dx = events[i].x - prev_x;
    area += dx * T.total_length(); // current union height
    if (events[i].type == START)
        T.insert(events[i].y1, events[i].y2);
    else
        T.remove(events[i].y1, events[i].y2);
    prev_x = events[i].x;
}
```

3. Other Applications

Problem	Description	Time
K-closest points	Maintain top k in active set	$O(n \log n)$
Union of rectangles	Compute covered area	$O(n \log n)$
Point location	Locate point in planar subdivision	$O(\log n)$
Visibility graph	Track visible edges	$O(n \log n)$

4. Plane Sweep Extensions

While line sweep moves in one dimension (x), plane sweep handles 2D or higher-dimensional spaces, where:

- Events are 2D cells or regions.- Sweep front is a plane instead of a line. Used in 3D collision detection, computational topology, and CAD systems.

Conceptual Visualization

1. Sort events by one axis (say, x).
2. Maintain structure (set, tree, or heap) of intersecting or active elements.
3. Update at each event and record desired output (intersection, union, coverage).

The key is the locality principle: only neighbors in the sweep structure can change outcomes.

5. Complexity

Phase	Complexity
Sorting events	$O(n \log n)$
Processing events	$O(n \log n)$
Total	$O(n \log n)$ (typical)

Why It Matters

The sweep line method transforms geometric chaos into order , turning spatial relationships into sorted sequences. It's the bridge between geometry and algorithms, blending structure with motion.

> "A sweep line sees everything , not all at once, but just in time."

Try It Yourself

1. Implement a sweep-line segment intersection finder.
2. Compute the union area of 3 rectangles with overlaps.
3. Animate the sweep line to visualize event processing.
4. Modify for circular or polygonal objects.
5. Explore how sweep-line logic applies to time-based events in scheduling.

74. Delaunay and Voronoi Diagrams

In geometry and spatial computing, Delaunay triangulations and Voronoi diagrams are duals , elegant structures that capture proximity, territory, and connectivity among points.

They're used everywhere: from mesh generation, pathfinding, geospatial analysis, to computational biology. This section introduces both, their relationship, and algorithms to construct them efficiently.

1. Voronoi Diagram

Given a set of sites (points) $P = p_1, p_2, \dots, p_n$, the Voronoi diagram partitions the plane into regions , one per point , so that every location in a region is closer to its site than to any other.

Formally, the Voronoi cell for p_i is:

$$V(p_i) = x \in \mathbb{R}^2 \mid d(x, p_i) \leq d(x, p_j), \forall j \neq i$$

Each region is convex, and boundaries are formed by perpendicular bisectors.

Example

For points (A, B, C):

- Draw bisectors between each pair.- Intersection points define Voronoi vertices.- Resulting polygons cover the plane, one per site. Used to model nearest neighbor regions , "which tower serves which area?"

Properties

- Every cell is convex.- Neighboring cells share edges.- The diagram's vertices are centers of circumcircles through three sites.- Dual graph = Delaunay triangulation.

2. Delaunay Triangulation

The Delaunay triangulation (DT) connects points so that no point lies inside the circumcircle of any triangle.

Equivalently, it's the dual graph of the Voronoi diagram.

It tends to avoid skinny triangles , maximizing minimum angles, creating well-shaped meshes.

Formal Definition

A triangulation (T) of (P) is Delaunay if for every triangle $\triangle abc \in T$, no point $p \in P \setminus a, b, c$ lies inside the circumcircle of $\triangle abc$.

Why It Matters:

- Avoids sliver triangles.- Used in finite element meshes, terrain modeling, and path planning.- Leads to natural neighbor interpolation and smooth surfaces.

3. Relationship

Voronoi and Delaunay are geometric duals:

Voronoi	Delaunay
Regions = proximity zones	Triangles = neighbor connections
Edges = bisectors	Edges = neighbor pairs
Vertices = circumcenters	Faces = circumcircles

Connecting neighboring Voronoi cells gives Delaunay edges.

4. Algorithms

Several algorithms can build these diagrams efficiently.

A. Incremental Insertion

1. Start with a super-triangle enclosing all points.
2. Insert points one by one.
3. Remove triangles whose circumcircle contains the point.
4. Re-triangulate the resulting polygonal hole.

Time Complexity: ($\mathrm{O}n^2$), improved to ($\mathrm{O}n \log n$) with randomization.

B. Divide and Conquer

1. Sort points by x.
2. Recursively build DT for left and right halves.
3. Merge by finding common tangents.

Time Complexity: ($\mathrm{O}n \log n$) Elegant, structured, and deterministic.

C. Fortune's Sweep Line Algorithm

For Voronoi diagrams, Fortune's algorithm sweeps a line from top to bottom. Maintains a beach line of parabolic arcs and event queue.

Each event (site or circle) updates the structure , building Voronoi edges incrementally.

Time Complexity: ($On \log n$)

D. Bowyer-Watson (Delaunay via Circumcircle Test)**

A practical incremental version widely used in graphics and simulation.

Steps:

- Start with supertriangle- Insert point- Remove all triangles whose circumcircle contains point- Reconnect the resulting cavity

Tiny Code (Conceptual)

```
typedef struct { double x, y; } Point;

typedef struct { Point a, b, c; } Triangle;

bool in_circle(Point a, Point b, Point c, Point p) {
    double A[3][3] = {
        {a.x - p.x, a.y - p.y, (a.x*a.x + a.y*a.y) - (p.x*p.x + p.y*p.y)},
        {b.x - p.x, b.y - p.y, (b.x*b.x + b.y*b.y) - (p.x*p.x + p.y*p.y)},
        {c.x - p.x, c.y - p.y, (c.x*c.x + c.y*c.y) - (p.x*p.x + p.y*p.y)}
    };
    return determinant(A) > 0;
}
```

This test ensures Delaunay property.

5. Applications

Domain	Application
GIS	Nearest facility, region partition
Mesh Generation	Finite element methods
Robotics	Visibility graphs, navigation
Computer Graphics	Terrain triangulation

Domain	Application
Clustering	Spatial neighbor structure

6. Complexity Summary

Algorithm	Type	Time	Notes
Fortune	Voronoi	($On \log n$)	Sweep line
Bowyer-Watson	Delaunay	($On \log n$)	Incremental
Divide & Conquer	Delaunay	($On \log n$)	Recursive

Why It Matters

Voronoi and Delaunay diagrams reveal natural structure in point sets. They convert distance into geometry, showing how space is divided and connected. If geometry is the shape of space, these diagrams are its skeleton.

> "Every point claims its territory; every territory shapes its network."

Try It Yourself

1. Draw Voronoi regions for 5 random points by hand.
2. Build Delaunay triangles (connect neighboring sites).
3. Verify the empty circumcircle property.
4. Use a library (CGAL / SciPy) to visualize both structures.
5. Explore how adding new points reshapes the diagrams.

75. Point in Polygon and Polygon Triangulation

Geometry often asks two fundamental questions:

1. Is a point inside or outside a polygon?
2. How can a complex polygon be broken into triangles for computation?

These are the building blocks of spatial analysis, computer graphics, and computational geometry.

1. Point in Polygon (PIP)

Given a polygon defined by vertices ($x_1, y_1, x_2, y_2, ..., x_n, y_n$) and a test point ((x, y)), we want to determine if the point lies inside, on the boundary, or outside the polygon.

Methods

A. Ray Casting Algorithm

Shoot a ray horizontally to the right of the point. Count how many times it intersects polygon edges.

- Odd count → Inside- Even count → Outside This is based on the even-odd rule.

Tiny Code (Ray Casting in C)

```c
bool point_in_polygon(Point p, Point poly[], int n) {
    bool inside = false;
    for (int i = 0, j = n - 1; i < n; j = i++) {
        if (((poly[i].y > p.y) != (poly[j].y > p.y)) &&
            (p.x < (poly[j].x - poly[i].x) *
                   (p.y - poly[i].y) /
                   (poly[j].y - poly[i].y) + poly[i].x))
            inside = !inside;
    }
    return inside;
}
```

This toggles `inside` every time a crossing is found.

B. Winding Number Algorithm

Counts how many times the polygon winds around the point.

- Nonzero winding number → Inside- Zero → Outside More robust for complex polygons with holes or self-intersections.

Method	Time Complexity	Robustness
Ray Casting	(O(n))	Simple, may fail on edge cases
Winding Number	(O(n))	More accurate for complex shapes

Edge Cases

Handle:

- Points on edges or vertices- Horizontal edges (special treatment to avoid double counting) Numerical precision is key.

Applications

- Hit testing in computer graphics- GIS spatial queries- Collision detection

2. Polygon Triangulation

A polygon triangulation divides a polygon into non-overlapping triangles whose union equals the polygon.

Why triangulate?

- Triangles are simple, stable, and efficient for rendering and computation.- Used in graphics pipelines, area computation, physics, and mesh generation.

A. Triangulation Basics

For a simple polygon with (n) vertices,

- Always possible- Always yields (n - 2) triangles Goal: Find a triangulation efficiently and stably.

B. Ear Clipping Algorithm

An intuitive and widely used method for triangulation.

Idea

1. Find an ear: a triangle formed by three consecutive vertices (v_{i-1}, v_i, v_{i+1}) such that:
   - It is convex - Contains no other vertex inside
2. Clip the ear (remove vertex v_i)
3. Repeat until only one triangle remains

Time Complexity: ($\mathrm{O}n^2$)

Tiny Code (Ear Clipping Sketch)

```
while (n > 3) {
    for (i = 0; i < n; i++) {
        if (is_ear(i)) {
            add_triangle(i-1, i, i+1);
            remove_vertex(i);
            break;
        }
    }
}
```

Helper `is_ear()` checks convexity and emptiness.

C. Dynamic Programming for Convex Polygons

If the polygon is convex, use DP triangulation:

$$dp[i][j] = \min_{k \in (i,j)} dp[i][k] + dp[k][j] + cost(i, j, k)$$

Cost: perimeter or area (for minimum-weight triangulation)

Time Complexity: ($\mathrm{O}n^3$) Space: ($\mathrm{O}n^2$)

D. Divide and Conquer

Recursively split polygon and triangulate sub-polygons. Useful for convex or near-convex shapes.

Algorithm	Time	Notes
Ear Clipping	$(\mathrm{O}n^2)$	Simple polygons
DP Triangulation	$(\mathrm{O}n^3)$	Weighted cost
Convex Polygon	(O(n))	Straightforward

3. Applications

Domain	Usage
Computer Graphics	Rendering, rasterization
Computational Geometry	Area computation, integration
Finite Element Analysis	Mesh subdivision
Robotics	Path planning, map decomposition

Why It Matters

Point-in-polygon answers where you are. Triangulation tells you how space is built. Together, they form the foundation of geometric reasoning.

> "From a single point to a thousand triangles, geometry turns space into structure."

Try It Yourself

1. Draw a non-convex polygon and test random points using the ray casting rule.
2. Implement the ear clipping algorithm for a simple polygon.
3. Visualize how each step removes an ear and simplifies the shape.
4. Compare triangulation results for convex vs concave shapes.

76. Spatial Data Structures (KD, R-tree)

When working with geometric data, points, rectangles, or polygons, efficient lookup and organization are crucial. Spatial data structures are designed to answer queries like:

- Which objects are near a given point?- Which shapes intersect a region?- What's the nearest neighbor? They form the backbone of computational geometry, computer graphics, GIS, and search systems.

1. Motivation

Brute force approaches that check every object have (O(n)) or worse performance. Spatial indexing structures, like KD-Trees and R-Trees, enable efficient range queries, nearest neighbor searches, and spatial joins.

2. KD-Tree (k-dimensional tree)

A KD-tree is a binary tree that recursively partitions space using axis-aligned hyperplanes.

Each node splits the data by one coordinate axis (x, y, z, …).

Structure

- Each node represents a point.- Each level splits by a different axis (x, y, x, y, …).- Left child contains points with smaller coordinate.- Right child contains larger coordinate.

Tiny Code (KD-tree Construction in 2D)

```
typedef struct {
    double x, y;
} Point;

int axis; // 0 for x, 1 for y

KDNode* build(Point points[], int n, int depth) {
    if (n == 0) return NULL;
    axis = depth % 2;
    int mid = n / 2;
    nth_element(points, points + mid, points + n, compare_by_axis);
    KDNode* node = new_node(points[mid]);
    node->left  = build(points, mid, depth + 1);
    node->right = build(points + mid + 1, n - mid - 1, depth + 1);
    return node;
}
```

Search Complexity:

- Average: ($O\log n$)- Worst-case: (O(n))

Queries

- Range query: Find points in a region.- Nearest neighbor: Search branches that might contain closer points.- K-nearest neighbors: Use priority queues.

Pros & Cons

Pros	Cons
Efficient for static data	Costly updates
Good for low dimensions	Degrades with high dimensions

Applications

- Nearest neighbor in ML- Collision detection- Clustering (e.g., k-means acceleration)

3. R-Tree (Rectangle Tree)

An R-tree is a height-balanced tree for rectangular bounding boxes. It's the spatial analog of a B-tree.

Idea

- Store objects or bounding boxes in leaf nodes.- Internal nodes store MBRs (Minimum Bounding Rectangles) that cover child boxes.- Query by traversing overlapping MBRs.

Tiny Code (R-Tree Node Sketch)

```
typedef struct {
    Rectangle mbr;
    Node* children[MAX_CHILDREN];
    int count;
} Node;
```

Insertion chooses the child whose MBR expands least to accommodate the new entry.

Operations

- Insert: Choose subtree → Insert → Adjust MBRs- Search: Descend into nodes whose MBR intersects query- Split: When full, use heuristics (linear, quadratic, R*-Tree) Complexity:
- Query: (O$\log n$)- Insert/Delete: (O$\log n$) average

Pros & Cons

Pros	Cons
Supports dynamic data	Overlaps can degrade performance
Ideal for rectangles	Complex split rules

Variants

- R*-Tree: Optimized reinsertion, better packing- R+ Tree: Non-overlapping partitions- Hilbert R-Tree: Uses space-filling curves

4. Comparison

Feature	KD-Tree	R-Tree
Data Type	Points	Rectangles / Regions
Dimensionality	Low (2-10)	Medium
Use Case	NN, range queries	Spatial joins, overlap queries
Updates	Expensive	Dynamic-friendly
Balance	Recursive median	B-tree-like

5. Other Spatial Structures

Structure	Description
Quadtree	Recursive 2D subdivision into 4 quadrants
Octree	3D analog of quadtree
BSP Tree	Binary partition using arbitrary planes
Grid Index	Divide space into uniform grid cells

6. Applications

Domain	Usage
GIS	Region queries, map intersections
Graphics	Ray tracing acceleration
Robotics	Collision and path planning
ML	Nearest neighbor search
Databases	Spatial indexing

Why It Matters

Spatial structures turn geometry into searchable data. They enable efficient algorithms for where and what's near, vital for real-time systems.

> "Divide space wisely, and queries become whispers instead of shouts."

Try It Yourself

1. Build a KD-tree for 10 random 2D points.
2. Implement nearest neighbor search.
3. Insert rectangles into a simple R-tree and query intersection with a bounding box.
4. Compare query time vs brute force.

77. Rasterization and Scanline Techniques

When you draw shapes on a screen, triangles, polygons, circles, they must be converted into pixels. This conversion is called rasterization. It's the bridge between geometric math and visible images.

Rasterization and scanline algorithms are foundational to computer graphics, game engines, and rendering pipelines.

1. What Is Rasterization?

Rasterization transforms vector shapes (continuous lines and surfaces) into discrete pixels on a grid.

For example, a triangle defined by vertices `(x1, y1), (x2, y2), (x3, y3)` must be filled pixel by pixel.

2. Core Idea

Each shape (line, polygon, circle) is sampled over a grid. The algorithm decides which pixels are inside, on, or outside the shape.

A rasterizer answers:

- Which pixels should be lit?- What color or depth should each pixel have?

3. Line Rasterization (Bresenham's Algorithm)

A classic method for drawing straight lines with integer arithmetic.

Key Idea: Move from one pixel to the next, choosing the pixel closest to the true line path.

```c
void draw_line(int x0, int y0, int x1, int y1) {
    int dx = abs(x1 - x0), dy = abs(y1 - y0);
    int sx = (x0 < x1) ? 1 : -1;
    int sy = (y0 < y1) ? 1 : -1;
    int err = dx - dy;
    while (true) {
        plot(x0, y0); // draw pixel
        if (x0 == x1 && y0 == y1) break;
        int e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x0 += sx; }
        if (e2 < dx) { err += dx; y0 += sy; }
    }
}
```

Why it works: Bresenham avoids floating-point math and keeps the line visually continuous.

4. Polygon Rasterization

To fill shapes, we need scanline algorithms, they sweep a horizontal line (y-axis) across the shape and fill pixels in between edges.

Scanline Fill Steps

1. Sort edges by their y-coordinates.
2. Scan each line (y).
3. Find intersections with polygon edges.
4. Fill between intersection pairs.

This guarantees correct filling for convex and concave polygons.

Example (Simple Triangle Rasterization)

```
for (int y = y_min; y <= y_max; y++) {
    find all x-intersections with polygon edges;
    sort x-intersections;
    for (int i = 0; i < count; i += 2)
        draw_line(x[i], y, x[i+1], y);
}
```

5. Circle Rasterization (Midpoint Algorithm)

Use symmetry, a circle is symmetric in 8 octants.

Each step calculates the error term to decide whether to move horizontally or diagonally.

```
void draw_circle(int xc, int yc, int r) {
    int x = 0, y = r, d = 3 - 2 * r;
    while (y >= x) {
        plot_circle_points(xc, yc, x, y);
        x++;
        if (d > 0) { y--; d += 4 * (x - y) + 10; }
        else d += 4 * x + 6;
    }
}
```

6. Depth and Shading

In 3D graphics, rasterization includes depth testing (Z-buffer) and color interpolation. Each pixel stores its depth; new pixels overwrite only if closer.

Interpolated shading (Gouraud, Phong) computes smooth color transitions across polygons.

7. Hardware Rasterization

Modern GPUs perform rasterization in parallel:

- Vertex Shader → Projection- Rasterizer → Pixel Grid- Fragment Shader → Color & Depth Each pixel is processed in fragment shaders for lighting, texture, and effects.

8. Optimizations

Technique	Purpose
Bounding Box Clipping	Skip off-screen regions
Early Z-Culling	Discard hidden pixels early
Edge Functions	Fast inside-test for triangles
Barycentric Coordinates	Interpolate depth/color smoothly

9. Why It Matters

Rasterization turns math into imagery. It's the foundation of all visual computing, renderers, CAD, games, and GUIs. Even with ray tracing rising, rasterization remains dominant for real-time rendering.

> "Every pixel you see began as math, it's just geometry painted by light."

10. Try It Yourself

1. Implement Bresenham's algorithm for lines.
2. Write a scanline polygon fill for triangles.
3. Extend it with color interpolation using barycentric coordinates.
4. Compare performance vs brute force (looping over all pixels).

78. Computer Vision (Canny, Hough, SIFT)

Computer vision is where algorithms learn to see, to extract structure, shape, and meaning from images. Behind every object detector, edge map, and keypoint matcher lies a handful of powerful geometric algorithms.

In this section, we explore four pillars of classical vision: Canny edge detection, Hough transform, and SIFT (Scale-Invariant Feature Transform).

1. The Vision Pipeline

Most vision algorithms follow a simple pattern:

1. Input: Raw pixels (grayscale or color)
2. Preprocess: Smoothing or filtering
3. Feature extraction: Edges, corners, blobs
4. Detection or matching: Shapes, keypoints
5. Interpretation: Object recognition, tracking

Canny, Hough, and SIFT live in the feature extraction and detection stages.

2. Canny Edge Detector

Edges mark places where intensity changes sharply, the outlines of objects. The Canny algorithm (1986) is one of the most robust and widely used edge detectors.

Steps

1. Smoothing: Apply Gaussian blur to reduce noise.
2. Gradient computation:
   - Compute G_x and G_y via Sobel filters
   - Gradient magnitude: $G = \sqrt{G_x^2 + G_y^2}$
   - Gradient direction: $\theta = \tan^{-1} \frac{G_y}{G_x}$
3. Non-maximum suppression:
   - Keep only local maxima along the gradient direction
4. Double thresholding:
   - Strong edges (high gradient)
   - Weak edges (connected to strong ones)
5. Edge tracking by hysteresis:
   - Connect weak edges linked to strong edges

Tiny Code (Pseudocode)

```
Image canny(Image input) {
    Image smoothed = gaussian_blur(input);
    Gradient grad = sobel(smoothed);
    Image suppressed = non_max_suppression(grad);
    Image edges = hysteresis_threshold(suppressed, low, high);
    return edges;
}
```

Why Canny Works

Canny maximizes three criteria:

1. Good detection (low false negatives)
2. Good localization (edges close to true edges)
3. Single response (no duplicates)

It's a careful balance between sensitivity and stability.

3. Hough Transform

Canny finds edge points, Hough connects them into shapes.

The Hough transform detects lines, circles, and other parametric shapes using voting in parameter space.

Line Detection

Equation of a line:

$$\rho = x\cos\theta + y\sin\theta$$

Each edge point votes for all (ρ, θ) combinations it could belong to. Peaks in the accumulator array correspond to strong lines.

Tiny Code (Hough Transform)

```
for each edge point (x, y):
  for theta in [0, 180):
    rho = x*cos(theta) + y*sin(theta);
    accumulator[rho, theta]++;
```

Then pick (ρ, θ) with highest votes.

Circle Detection

Use 3D accumulator $center_x, center_y, radius$. Each edge pixel votes for possible circle centers.

Applications

- Lane detection in self-driving- Shape recognition (circles, ellipses)- Document analysis (lines, grids)

4. SIFT (Scale-Invariant Feature Transform)

SIFT finds keypoints that remain stable under scale, rotation, and illumination changes.

It's widely used for image matching, panoramas, 3D reconstruction, and object recognition.

Steps

1. Scale-space extrema detection

   - Use Difference of Gaussians (DoG) across scales. - Detect maxima/minima in space-scale neighborhood.2. Keypoint localization
   - Refine keypoint position and discard unstable ones.3. Orientation assignment
   - Assign dominant gradient direction.4. Descriptor generation
   - Build a 128D histogram of gradient orientations in a local patch.

Tiny Code (Outline)

```
for each octave:
  build scale-space pyramid
  find DoG extrema
  localize keypoints
  assign orientations
  compute 128D descriptor
```

Properties

Property	Description
Scale Invariant	Detects features at multiple scales
Rotation Invariant	Uses local orientation
Robust	Handles lighting, noise, affine transforms

5. Comparison

Algorithm	Purpose	Output	Robustness
Canny	Edge detection	Binary edge map	Sensitive to thresholds
Hough	Shape detection	Lines, circles	Needs clean edges
SIFT	Feature detection	Keypoints, descriptors	Very robust

6. Applications

Domain	Use Case
Robotics	Visual SLAM, localization
AR / VR	Marker tracking
Search	Image matching
Medical	Edge segmentation
Industry	Quality inspection

7. Modern Successors

- ORB (FAST + BRIEF): Efficient for real-time- SURF: Faster SIFT alternative- Harris / FAST: Corner detectors- Deep features: CNN-based descriptors

Why It Matters

These algorithms gave machines their first eyes, before deep learning, they were how computers recognized structure. Even today, they're used in preprocessing, embedded systems, and hybrid pipelines.

"Before neural nets could dream, vision began with gradients, geometry, and votes."

Try It Yourself

1. Implement Canny using Sobel and hysteresis.
2. Use Hough transform to detect lines in a synthetic image.
3. Try OpenCV SIFT to match keypoints between two rotated images.
4. Compare edge maps before and after Gaussian blur.

79. Pathfinding in Space (A*, RRT, PRM)

When navigating a maze, driving an autonomous car, or moving a robot arm, the question is the same: How do we find a path from start to goal efficiently and safely?

Pathfinding algorithms answer this question, balancing optimality, speed, and adaptability. In this section, we explore three foundational families:

- A*: Heuristic search in grids and graphs- RRT (Rapidly-Exploring Random Tree): Sampling-based exploration- PRM (Probabilistic Roadmap): Precomputed navigation networks

1. The Pathfinding Problem

Given:

- A space (grid, graph, or continuous)- A start node and goal node- A cost function (distance, time, energy)- Optional obstacles Find a collision-free, low-cost path.

2. A* (A-star) Search

A* combines Dijkstra's algorithm with a heuristic that estimates remaining cost. It's the most popular graph-based pathfinding algorithm.

Key Idea

Each node (n) has:

$$f(n) = g(n) + h(n)$$

- (g(n)): cost so far- (h(n)): estimated cost to goal- (f(n)): total estimated cost

Algorithm

1. Initialize priority queue with start node

2. While queue not empty:

   - Pop node with smallest (f(n)) - If goal reached → return path - For each neighbor:
     - Compute new (g), (f) - Update queue if better

Tiny Code (Grid A*)

```
typedef struct { int x, y; double g, f; } Node;

double heuristic(Node a, Node b) {
    return fabs(a.x - b.x) + fabs(a.y - b.y); // Manhattan
}

void a_star(Node start, Node goal) {
    PriorityQueue open;
    push(open, start);
    while (!empty(open)) {
        Node cur = pop_min(open);
        if (cur == goal) return reconstruct_path();
        for (Node n : neighbors(cur)) {
            double tentative_g = cur.g + dist(cur, n);
            if (tentative_g < n.g) {
                n.g = tentative_g;
                n.f = n.g + heuristic(n, goal);
                push(open, n);
            }
        }
    }
}
```

Complexity

- Time: ($OE\log V$)- Space: (O(V))- Optimal if (h(n)) is admissible (never overestimates)

Variants

Variant	Description
Dijkstra	A* with (h(n) = 0)
Greedy Best-First	Uses (h(n)) only
Weighted A*	Speeds up with tradeoff on optimality

Variant	Description
Jump Point Search	Optimized for uniform grids

3. RRT (Rapidly-Exploring Random Tree)

A* struggles in continuous or high-dimensional spaces (e.g. robot arms). RRT tackles this with randomized exploration.

Core Idea

- Grow a tree from the start by randomly sampling points.- Extend tree toward each sample (step size ϵ).- Stop when near the goal.

Tiny Code (RRT Sketch)

```
Tree T = {start};
for (int i = 0; i < MAX_ITERS; i++) {
    Point q_rand = random_point();
    Point q_near = nearest(T, q_rand);
    Point q_new = steer(q_near, q_rand, step_size);
    if (collision_free(q_near, q_new))
        add_edge(T, q_near, q_new);
    if (distance(q_new, goal) < eps)
        return path;
}
```

Pros & Cons

Pros	Cons
Works in continuous space	Paths are suboptimal
Handles high dimensions	Randomness may miss narrow passages
Simple and fast	Needs post-processing (smoothing)

Variants

Variant	Description
RRT*	Asymptotically optimal
Bi-RRT	Grow from both start and goal
Informed RRT*	Focus on promising regions

4. PRM (Probabilistic Roadmap)

PRM builds a graph of feasible configurations, a roadmap, then searches it.

Steps

1. Sample random points in free space
2. Connect nearby points with collision-free edges
3. Search roadmap (e.g., with A*)

Tiny Code (PRM Sketch)

```
Graph G = {};
for (int i = 0; i < N; i++) {
    Point p = random_free_point();
    G.add_vertex(p);
}
for each p in G:
    for each q near p:
        if (collision_free(p, q))
            G.add_edge(p, q);
path = a_star(G, start, goal);
```

Pros & Cons

Pros	Cons
Precomputes reusable roadmap	Needs many samples for coverage
Good for multiple queries	Poor for single-query planning
Works in high-dim spaces	May need post-smoothing

5. Comparison

Algorithm	Space	Nature	Optimal	Use Case
A*	Discrete	Deterministic	Yes	Grids, graphs
RRT	Continuous	Randomized	No (RRT* = Yes)	Robotics, motion planning
PRM	Continuous	Randomized	Approx.	Multi-query planning

6. Applications

Domain	Use Case
Robotics	Arm motion, mobile navigation
Games	NPC pathfinding, AI navigation mesh
Autonomous vehicles	Route planning
Aerospace	Drone and spacecraft trajectory
Logistics	Warehouse robot movement

Why It Matters

Pathfinding is decision-making in space, it gives agents the ability to move, explore, and act purposefully. From Pac-Man to Mars rovers, every journey starts with an algorithm.

"To move with purpose, one must first see the paths that are possible."

Try It Yourself

1. Implement A* on a 2D grid with walls.
2. Generate an RRT in a 2D obstacle field.
3. Build a PRM for a continuous space and run A* on the roadmap.
4. Compare speed and path smoothness across methods.

80. Computational Geometry Variants and Applications

Computational geometry is the study of algorithms on geometric data, points, lines, polygons, circles, and higher-dimensional shapes. By now, you've seen core building blocks: convex hulls, intersections, nearest neighbors, triangulations, and spatial indexing.

This final section brings them together through variants, generalizations, and real-world applications, showing how geometry quietly powers modern computing.

1. Beyond the Plane

Most examples so far assumed 2D geometry. But real systems often live in 3D or N-D spaces.

Dimension	Example Problems	Typical Uses
2D	Convex hull, polygon area, line sweep	GIS, CAD, mapping
3D	Convex polyhedra, mesh intersection, visibility	Graphics, simulation
N-D	Voronoi in high-D, KD-trees, optimization	ML, robotics, data science

Higher dimensions add complexity (and sometimes impossibility):

- Exact geometry often replaced by approximations.- Volume, distance, and intersection tests become more expensive.

2. Approximate and Robust Geometry

Real-world geometry faces numerical errors (floating point) and degenerate cases (collinear, overlapping). To handle this, algorithms adopt robustness and approximation strategies.

- Epsilon comparisons: treat values within tolerance as equal- Orientation tests: robustly compute turn direction via cross product- Exact arithmetic: rational or symbolic computation- Grid snapping: quantize space for stability Approximate geometry accepts small error for large speed-up, essential in graphics and machine learning.

3. Geometric Duality

A powerful tool for reasoning about problems: map points to lines, lines to points. For example:

- A point ((a, b)) maps to line (y = ax - b).- A line (y = mx + c) maps to point ((m, -c)). Applications:
- Transforming line intersection problems into point location problems- Simplifying half-plane intersections- Enabling arrangement algorithms in computational geometry Duality is a common trick: turn geometry upside-down to make it simpler.

4. Geometric Data Structures

Recap of core spatial structures and what they're best at:

Structure	Stores	Queries	Use Case
KD-Tree	Points	NN, range	Low-D search
R-Tree	Rectangles	Overlaps	Spatial DB
Quad/Octree	Space partitions	Point lookup	Graphics, GIS
BSP Tree	Polygons	Visibility	Rendering
Delaunay Triangulation	Points	Neighbors	Mesh generation
Segment Tree	Intervals	Range	Sweep-line events

5. Randomized Geometry

Randomness simplifies deterministic geometry:

- Randomized incremental construction (Convex Hulls, Delaunay)- Random sampling for approximation (-nets, VC dimension)- Monte Carlo geometry for probabilistic intersection and coverage Example: randomized incremental convex hull builds expected ($On \log n$) structures with elegant proofs.

6. Computational Topology

Beyond geometry lies shape connectivity, studied by topology. Algorithms compute connected components, holes, homology, and Betti numbers.

Applications include:

- 3D printing (watertightness)- Data analysis (persistent homology)- Robotics (free space topology) Geometry meets topology in alpha-shapes, simplicial complexes, and manifold reconstruction.

7. Geometry Meets Machine Learning

Many ML methods are geometric at heart:

- Nearest neighbor $\rightarrow$ Voronoi diagram- SVM $\rightarrow$ hyperplane separation- K-means $\rightarrow$ Voronoi partitioning- Manifold learning $\rightarrow$ low-dim geometry- Convex optimization $\rightarrow$ geometric feasibility Visualization tools (t-SNE, UMAP) rely on spatial embedding and distance geometry.

8. Applications Across Fields

Field	Application	Geometric Core
Graphics	Rendering, collision	Triangulation, ray tracing
GIS	Maps, roads	Polygons, point-in-region
Robotics	Path planning	Obstacles, configuration space
Architecture	Modeling	Mesh operations
Vision	Object boundaries	Contours, convexity
AI	Clustering, similarity	Distance metrics
Physics	Simulation	Particle collision
Databases	Spatial joins	R-Trees, indexing

Geometry underpins structure, position, and relationship, the backbone of spatial reasoning.

9. Complexity and Open Problems

Some problems still challenge efficient solutions:

- Point location in dynamic settings- Visibility graphs in complex polygons- Motion planning in high dimensions- Geometric median / center problems- Approximation guarantees in robust settings These remain active areas in computational geometry research.

Tiny Code (Point-in-Polygon via Ray Casting)

```
bool inside(Point p, Polygon poly) {
    int cnt = 0;
    for (int i = 0; i < poly.n; i++) {
        Point a = poly[i], b = poly[(i + 1) % poly.n];
        if (intersect_ray(p, a, b)) cnt++;
    }
    return cnt % 2 == 1; // odd crossings = inside
}
```

This small routine appears everywhere, maps, games, GUIs, and physics engines.

10. Why It Matters

Computational geometry is more than shape, it's the mathematics of space, powering visual computing, spatial data, and intelligent systems. Everywhere something moves, collides, maps, or recognizes form, geometry is the invisible hand guiding it.

> "All computation lives somewhere, and geometry is how we understand the where."

Try It Yourself

1. Implement point-in-polygon and test on convex vs concave shapes.
2. Visualize a Delaunay triangulation and its Voronoi dual.
3. Experiment with KD-trees for nearest neighbor queries.
4. Write a small convex hull in 3D using incremental insertion.
5. Sketch an RRT path over a geometric map.

Chapter 9. Systems, Databases, and Distributed Algorithms

81. Concurrency Control (2PL, MVCC, OCC)

In multi-user or multi-threaded systems, many operations want to read or write shared data at the same time. Without discipline, this leads to chaos, lost updates, dirty reads, or even inconsistent states.

Concurrency control ensures correctness under parallelism, so that the result is as if each transaction ran alone (a property called serializability).

This section introduces three foundational techniques:

- 2PL - Two-Phase Locking- MVCC - Multi-Version Concurrency Control- OCC - Optimistic Concurrency Control

1. The Goal: Serializability

We want transactions to behave as if executed in some serial order, even though they're interleaved.

A schedule is *serializable* if it yields the same result as some serial order of transactions.

Concurrency control prevents problems like:

- Lost Update: Two writes overwrite each other.- Dirty Read: Read uncommitted data.- Non-repeatable Read: Data changes mid-transaction.- Phantom Read: New rows appear after a query.

2. Two-Phase Locking (2PL)

Idea: Use locks to coordinate access. Each transaction has two phases:

1. Growing phase: acquire locks (shared or exclusive)
2. Shrinking phase: release locks (no new locks allowed after release)

This ensures conflict-serializability.

Lock Types

Type	Operation	Shared?	Exclusive?
Shared (S)	Read	Yes	No
Exclusive (X)	Write	No	No

If a transaction needs to read: request S-lock. If it needs to write: request X-lock.

Tiny Code (Lock Manager Sketch)

```
void acquire_lock(Transaction *T, Item *X, LockType type) {
    while (conflict_exists(X, type))
        wait();
    add_lock(X, T, type);
}

void release_all(Transaction *T) {
    for (Lock *l in T->locks)
        unlock(l);
}
```

Example

```
T1: read(A); write(A)
T2: read(A); write(A)
```

Without locks → race condition. With 2PL → one must wait → consistent.

Variants

Variant	Description
Strict 2PL	Holds all locks until commit → avoids cascading aborts
Rigorous 2PL	Same as Strict, all locks released at end
Conservative 2PL	Acquires all locks before execution

Pros & Cons

Pros	Cons
Guarantees serializability	Can cause deadlocks
Simple concept	Blocking, contention under load

3. Multi-Version Concurrency Control (MVCC)

Idea: Readers don't block writers, and writers don't block readers. Each write creates a new version of data with a timestamp.

Transactions read from a consistent snapshot based on their start time.

Snapshot Isolation

- Readers see the latest committed version at transaction start.- Writers produce new versions; conflicts detected at commit time. Each record stores:
- `value`- `created_at`- `deleted_at` (if applicable)

Tiny Code (Version Chain)

```
struct Version {
    int value;
    Timestamp created;
    Timestamp deleted;
    Version *next;
};
```

Read finds version with `created <= tx.start && deleted > tx.start`.

Pros & Cons

Pros	Cons
No read locks	Higher memory (multiple versions)
Readers never block	Write conflicts at commit
Great for OLTP systems	GC of old versions needed

Used In

- PostgreSQL- Oracle- MySQL (InnoDB)- Spanner

4. Optimistic Concurrency Control (OCC)

Idea: Assume conflicts are rare. Let transactions run without locks. At commit time, validate, if conflicts exist, rollback.

Phases

1. Read phase - execute, read data, buffer writes.
2. Validation phase - check if conflicts occurred.
3. Write phase - apply changes if valid, else abort.

Tiny Code (OCC Validation)

```
bool validate(Transaction *T) {
    for (Transaction *U in committed_since(T.start))
        if (conflict(T, U))
            return false;
    return true;
}
```

Pros & Cons

Pros	Cons
No locks → no deadlocks	High abort rate under contention
Great for low-conflict workloads	Wasted work on abort

Used In

- In-memory DBs- Distributed systems- STM (Software Transactional Memory)

5. Choosing a Strategy

System Type	Preferred Control
OLTP (many reads/writes)	MVCC
OLAP (read-heavy)	MVCC or OCC
Real-time systems	2PL (predictable)
Low contention	OCC
High contention	2PL / MVCC

6. Why It Matters

Concurrency control is the backbone of consistency in databases, distributed systems, and even multi-threaded programs. It enforces correctness amid chaos, ensuring your data isn't silently corrupted.

> "Without order, parallelism is noise. Concurrency control is its conductor."

Try It Yourself

1. Simulate 2PL with two transactions updating shared data.
2. Implement a toy MVCC table with version chains.
3. Write an OCC validator for three concurrent transactions.
4. Experiment: under high conflict, which model performs best?

82. Logging, Recovery, and Commit Protocols

No matter how elegant your algorithms or how fast your storage, failures happen. Power cuts, crashes, and network splits are inevitable. What matters is recovery, restoring the system to a consistent state without losing committed work.

Logging, recovery, and commit protocols form the backbone of reliable transactional systems, ensuring durability and correctness in the face of crashes.

1. The Problem

We need to guarantee the ACID properties:

- Atomicity - all or nothing- Consistency - valid before and after- Isolation - no interference- Durability - once committed, always safe If a crash occurs mid-transaction, how do we roll back or redo correctly?

The answer: Log everything, then replay or undo after failure.

2. Write-Ahead Logging (WAL)

The golden rule:

> "Write log entries before modifying the database."

Every action is recorded in a sequential log on disk, ensuring the system can reconstruct the state.

Log Record Format

Each log entry typically includes:

- `LSN` (Log Sequence Number)- `Transaction ID`- `Operation` (update, insert, delete)- `Before image` (old value)- `After image` (new value)

```
struct LogEntry {
    int lsn;
    int tx_id;
    char op[10];
    Value before, after;
};
```

When a transaction commits, the system first flushes logs to disk (`fsync`). Only then is the commit acknowledged.

3. Recovery Actions

When the system restarts, it reads logs and applies a recovery algorithm.

Three Phases (ARIES Model)

1. Analysis - determine state at crash (active vs committed)
2. Redo - repeat all actions from last checkpoint
3. Undo - rollback incomplete transactions

ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) is the most widely used approach (IBM DB2, PostgreSQL, SQL Server).

Redo Rule

If the system committed before crash → redo all updates so data is preserved.

Undo Rule

If the system didn't commit → undo to maintain atomicity.

Tiny Code (Simplified Recovery Sketch)

```
void recover(Log log) {
    for (Entry e : log) {
        if (e.committed)
            apply(e.after);
        else
            apply(e.before);
    }
}
```

4. Checkpointing

Instead of replaying the entire log, systems take checkpoints, periodic snapshots marking a safe state.

Type	Description
Sharp checkpoint	Stop all transactions briefly, flush data + log
Fuzzy checkpoint	Mark consistent LSN; continue running

Checkpoints reduce recovery time: only replay after the last checkpoint.

5. Commit Protocols

In distributed systems, multiple nodes must agree to commit or abort together. This is handled by atomic commit protocols.

Two-Phase Commit (2PC)

Goal: All participants either commit or abort in unison.

Steps:

1. Prepare phase (voting):

    - Coordinator asks all participants to "prepare" - Each replies yes/no2. Commit phase (decision):

    - If all say yes → commit - Else → abort

```
Coordinator: PREPARE
Participants: VOTE YES / NO
Coordinator: COMMIT / ABORT
```

If the coordinator crashes after prepare, participants must wait → blocking protocol.

Tiny Code (2PC Pseudocode)

```
bool two_phase_commit(Participants P) {
    for each p in P:
        if (!p.prepare()) return abort_all();
    for each p in P:
        p.commit();
    return true;
}
```

Three-Phase Commit (3PC)

Improves on 2PC by adding an intermediate phase to avoid indefinite blocking. More complex, used in systems with reliable failure detection.

6. Logging in Distributed Systems

Each participant maintains its own WAL. To recover globally:

- Use coordinated checkpoints- Maintain global commit logs- Consensus-based protocols (Paxos Commit, Raft) can replace 2PC for high availability

7. Example Timeline

Step	Action
T1 updates record A	WAL entry written
T1 updates record B	WAL entry written
T1 commits	WAL flush, commit record
Crash!	Disk may be inconsistent
Restart	Recovery scans log, redoes T1

8. Pros and Cons

Approach	Strength	Weakness
WAL	Simple, durable	Write overhead
Checkpointing	Faster recovery	I/O spikes
2PC	Global atomicity	Blocking
3PC / Consensus	Non-blocking	Complex, slower

9. Real Systems

System	Strategy
PostgreSQL	WAL + ARIES + Checkpoint
MySQL (InnoDB)	WAL + Fuzzy checkpoint
Spanner	WAL + 2PC + TrueTime
Kafka	WAL for durability
RocksDB	WAL + LSM checkpoints

10. Why It Matters

Logging and commit protocols make data survive crashes and stay consistent across machines. Without them, every failure risks corruption.

> "Persistence is not about never failing, it's about remembering how to stand back up."

Try It Yourself

1. Write a toy WAL system that logs before writes.
2. Simulate a crash mid-transaction and replay the log.
3. Implement a simple 2PC coordinator with two participants.

4. Compare recovery time with vs without checkpoints.

83. Scheduling (Round Robin, EDF, Rate-Monotonic)

In operating systems and real-time systems, scheduling determines the order in which tasks or processes run. Since resources like CPU time are limited, a good scheduler aims to balance fairness, efficiency, and responsiveness.

1. The Goal of Scheduling

Every system has tasks competing for the CPU. Scheduling decides:

- Which task runs next- How long it runs- When it yields or preempts Different goals apply in different domains:

Domain	Objective
General-purpose OS	Fairness, responsiveness
Real-time systems	Meeting deadlines
Embedded systems	Predictability
High-performance servers	Throughput, latency balance

A scheduler's policy can be preemptive (interrupts tasks) or non-preemptive (waits for voluntary yield).

2. Round Robin Scheduling

Round Robin (RR) is one of the simplest preemptive schedulers. Each process gets a fixed time slice (quantum) and runs in a circular queue.

If a process doesn't finish, it's put back at the end of the queue.

Tiny Code: Round Robin (Pseudocode)

```
queue processes;
while (!empty(processes)) {
    process = dequeue(processes);
    run_for_quantum(process);
    if (!process.finished)
```

```
        enqueue(processes, process);
}
```

Characteristics

- Fair: Every process gets CPU time.- Responsive: Short tasks don't starve.- Downside: Context switching overhead if quantum is too small. #### Example

Process	Burst Time
P1	4
P2	3
P3	2

Quantum = 1 Order: P1, P2, P3, P1, P2, P3, P1, P2 → all finish fairly.

3. Priority Scheduling

Each task has a priority. The scheduler always picks the highest-priority ready task.

- Preemptive: A higher-priority task can interrupt a lower one.- Non-preemptive: The CPU is released voluntarily. #### Problems
- Starvation: Low-priority tasks may never run.- Solution: Aging - gradually increase waiting task priority.

4. Earliest Deadline First (EDF)

EDF is a dynamic priority scheduler for real-time systems. Each task has a deadline, and the task with the earliest deadline runs first.

Rule

At any time, run the ready task with the closest deadline.

Example

Task	Execution Time	Deadline
T1	1	3
T2	2	5
T3	1	2

Order: T3 $\rightarrow$ T1 $\rightarrow$ T2

EDF is optimal for preemptive scheduling of independent tasks on a single processor.

5. Rate-Monotonic Scheduling (RMS)

In periodic real-time systems, tasks repeat at fixed intervals. RMS assigns higher priority to tasks with shorter periods.

Task	Period	Priority
T1	2 ms	High
T2	5 ms	Medium
T3	10 ms	Low

It's static (priorities don't change) and optimal among fixed-priority schedulers.

Utilization Bound

For n tasks, RMS is guaranteed schedulable if:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

For example, for 3 tasks, $U \leq 0.78$.

6. Shortest Job First (SJF)

Run the task with the shortest burst time first.

- Non-preemptive SJF: Once started, runs to completion.- Preemptive SJF (Shortest Remaining Time First): Preempts if a shorter job arrives. Advantage: Minimizes average waiting time. Disadvantage: Needs knowledge of future job lengths.

7. Multilevel Queue Scheduling

Divide processes into classes (interactive, batch, system). Each class has its own queue with own policy, e.g.:

- Queue 1: System → RR (quantum = 10ms)- Queue 2: Interactive → RR (quantum = 50ms)- Queue 3: Batch → FCFS (First-Come-First-Serve) CPU is assigned based on queue priority.

8. Multilevel Feedback Queue (MLFQ)

Processes move between queues based on behavior.

- CPU-bound → move down (lower priority)- I/O-bound → move up (higher priority) Goal: Adaptive scheduling that rewards interactive tasks.

Used in modern OS kernels (Linux, Windows).

9. Scheduling Metrics

Metric	Meaning
Turnaround Time	Completion − Arrival
Waiting Time	Time spent in ready queue
Response Time	Time from arrival to first execution
Throughput	Completed tasks per unit time
CPU Utilization	% of time CPU is busy

Schedulers balance these based on design goals.

10. Why It Matters

Schedulers shape how responsive, efficient, and fair a system feels. In operating systems, they govern multitasking. In real-time systems, they ensure deadlines are met. In servers, they keep latency low and throughput high.

> "Scheduling is not just about time. It's about fairness, foresight, and flow."

Try It Yourself

1. Simulate Round Robin with quantum = 2, compare average waiting time.
2. Implement EDF for a set of periodic tasks with deadlines.
3. Check schedulability under RMS for 3 periodic tasks.
4. Explore Linux CFS (Completely Fair Scheduler) source code.
5. Compare SJF and RR for CPU-bound vs I/O-bound workloads.

84. Caching and Replacement (LRU, LFU, CLOCK)

Caching is the art of remembering the past to speed up the future. In computing, caches store recently used or frequently accessed data to reduce latency and load on slower storage (like disks or networks). The challenge: caches have limited capacity, so when full, we must decide what to evict. That's where replacement policies come in.

1. The Need for Caching

Caches appear everywhere:

- CPU: L1, L2, L3 caches speed up memory access- Databases: query results or index pages- Web browsers / CDNs: recently fetched pages- Operating systems: page cache for disk blocks The principle guiding all caches is locality:
- Temporal locality: recently used items are likely used again soon- Spatial locality: nearby items are likely needed next

2. Cache Replacement Problem

When the cache is full, which item should we remove?

We want to minimize cache misses (requests not found in cache).

Formally:

> Given a sequence of accesses, find a replacement policy that minimizes misses.

Theoretical optimal policy (OPT): always evict the item used farthest in the future. But OPT requires future knowledge, so we rely on heuristics like LRU, LFU, CLOCK.

3. Least Recently Used (LRU)

LRU evicts the least recently accessed item. It assumes recently used = likely to be used again.

Implementation Approaches

- Stack (list): move item to top on access- Hash map + doubly linked list: `O(1)` insert, delete, lookup #### Tiny Code: LRU (Simplified)

```
typedef struct Node {
    int key;
    struct Node *prev, *next;
} Node;

HashMap cache;
List lru_list;

void access(int key) {
    if (in_cache(key)) move_to_front(key);
    else {
        if (cache_full()) remove_lru();
        insert_front(key);
    }
}
```

Pros

- Good for workloads with strong temporal locality #### Cons
- Costly in hardware or massive caches (metadata overhead)

4. Least Frequently Used (LFU)

LFU evicts the least frequently accessed item.

Tracks usage count for each item:

- Increment on each access- Evict lowest-count item #### Example

Item	Accesses	Frequency
A	3	3
B	1	1
C	2	2

Evict B.

Variants

- LFU with aging: gradually reduce counts to adapt to new trends- Approximate LFU: counters in ranges (for memory efficiency) #### Pros
- Great for stable, repetitive workloads #### Cons
- Poor for workloads with shifting popularity (slow adaptation)

5. FIFO (First In First Out)

Simple but naive:

- Evict the oldest item, ignoring usage Used in simple hardware caches. Good when access pattern is cyclic, bad otherwise.

6. Random Replacement (RR)

Evict a random entry.

Surprisingly competitive in some high-concurrency systems, and trivial to implement. Used in memcached (as an option).

7. CLOCK Algorithm

A practical approximation of LRU, widely used in OS page replacement.

Each page has a reference bit (R). Pages form a circular list.

Algorithm:

1. Clock hand sweeps over pages.
2. If `R = 0`, evict page.
3. If `R = 1`, set `R = 0` and skip.

This mimics LRU with O(1) cost and low overhead.

8. Second-Chance and Enhanced CLOCK

Second-Chance: give recently used pages a "second chance" before eviction. Enhanced CLOCK: also uses modify bit (M) to prefer clean pages.

Used in Linux's page replacement (with Active/Inactive lists).

9. Adaptive Algorithms

Modern systems use hybrid or adaptive policies:

- ARC (Adaptive Replacement Cache) - balances recency and frequency- CAR (Clock with Adaptive Replacement) - CLOCK-style adaptation- TinyLFU - frequency sketch + admission policy- Hyperbolic caching - popularity decay for large-scale systems These adapt dynamically to changing workloads.

10. Why It Matters

Caching is the backbone of system speed:

- OS uses it for paging- Databases for buffer pools- CPUs for memory hierarchies- CDNs for global acceleration Choosing the right eviction policy can mean orders of magnitude improvement in latency and throughput.

  "A good cache remembers what matters, and forgets what no longer does."

Try It Yourself

1. Simulate a cache of size 3 with sequence: A B C A B D A B C D Compare LRU, LFU, and FIFO miss counts.
2. Implement LRU with a doubly-linked list and hash map in C.
3. Try CLOCK with reference bits, simulate a sweep.
4. Experiment with ARC and TinyLFU for dynamic workloads.
5. Measure hit ratios for different access patterns (sequential, random, looping).

85. Networking (Routing, Congestion Control)

Networking algorithms make sure data finds its way through vast, connected systems, efficiently, reliably, and fairly. Two core pillars of network algorithms are routing (deciding *where* packets go) and congestion control (deciding *how fast* to send them).

Together, they ensure the internet functions under heavy load, dynamic topology, and unpredictable demand.

1. The Goals of Networking Algorithms

- Correctness: all destinations are reachable if paths exist- Efficiency: use minimal resources (bandwidth, latency, hops)- Scalability: support large, dynamic networks- Robustness: recover from failures- Fairness: avoid starving flows

2. Types of Routing

Routing decides paths packets should follow through a graph-like network.

Static vs Dynamic Routing

- Static: fixed routes, manual configuration (good for small networks)- Dynamic: routes adjust automatically as topology changes (internet-scale) #### Unicast, Multicast, Broadcast
- Unicast: one-to-one (most traffic)- Multicast: one-to-many (video streaming, gaming)- Broadcast: one-to-all (local networks)

3. Shortest Path Routing

Most routing relies on shortest path algorithms:

Dijkstra's Algorithm

- Builds shortest paths from one source- Complexity: `O(E log V)` with priority queue Used in:
- OSPF (Open Shortest Path First)- IS-IS (Intermediate System to Intermediate System) #### Bellman-Ford Algorithm
- Handles negative edges- Basis for Distance-Vector routing (RIP) #### Tiny Code: Dijkstra for Routing

```
#define INF 1e9
int dist[MAX], visited[MAX];
vector<pair<int,int>> adj[MAX];

void dijkstra(int s, int n) {
    for (int i = 0; i < n; i++) dist[i] = INF;
    dist[s] = 0;
    priority_queue<pair<int,int>> pq;
    pq.push({0, s});
    while (!pq.empty()) {
        int u = pq.top().second; pq.pop();
        if (visited[u]) continue;
        visited[u] = 1;
        for (auto [v, w]: adj[u]) {
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                pq.push({-dist[v], v});
            }
        }
    }
}
```

4. Distance-Vector vs Link-State

Feature	Distance-Vector (RIP)	Link-State (OSPF)
Info Shared	Distance to neighbors	Full topology map
Convergence	Slower (loops possible)	Fast (SPF computation)
Complexity	Lower	Higher
Examples	RIP, BGP (conceptually)	OSPF, IS-IS

RIP uses Bellman-Ford. OSPF floods link-state updates, runs Dijkstra at each node.

5. Hierarchical Routing

Large-scale networks (like the Internet) use hierarchical routing:

- Routers grouped into Autonomous Systems (AS)- Intra-AS routing: OSPF, IS-IS- Inter-AS routing: BGP (Border Gateway Protocol) BGP exchanges reachability info, not shortest paths, and prefers policy-based routing (e.g., cost, contracts, peering).

6. Congestion Control

Even with good routes, we can't flood links. Congestion control ensures fair and efficient use of bandwidth.

Implemented primarily at the transport layer (TCP).

TCP Congestion Control

Key components:

- Additive Increase, Multiplicative Decrease (AIMD)- Slow Start: probe capacity- Congestion Avoidance: grow cautiously- Fast Retransmit / Recovery Modern variants:
- TCP Reno: classic AIMD- TCP Cubic: non-linear growth for high-speed networks- BBR (Bottleneck Bandwidth + RTT): model-based control #### Algorithm Sketch (AIMD)

```
On ACK: cwnd += 1/cwnd  // increase slowly
On loss: cwnd /= 2      // halve window
```

7. Queue Management

Routers maintain queues. Too full? => Packet loss, latency spikes, tail drop.

Solutions:

- RED (Random Early Detection) - drop packets early- CoDel (Controlled Delay) - monitor queue delay, drop adaptively These prevent bufferbloat, improving latency for real-time traffic.

8. Flow Control vs Congestion Control

- Flow Control: prevent sender from overwhelming receiver- Congestion Control: prevent sender from overwhelming network TCP uses both: receive window (rwnd) and congestion window (cwnd). Actual sending rate = `min(rwnd, cwnd)`.

9. Data Plane vs Control Plane

- Control Plane: decides routes (OSPF, BGP)- Data Plane: forwards packets (fast path) Modern networking (e.g. SDN, Software Defined Networking) separates these:
- Controller computes routes- Switches act on flow rules

10. Why It Matters

Routing and congestion control shape the performance of:

- The Internet backbone- Data center networks (with load balancing)- Cloud services and microservice meshes- Content delivery networks (CDNs) Every packet's journey, from your laptop to a global data center, relies on these ideas.

  "Networking is not magic. It's algorithms moving data through time and space."

Try It Yourself

1. Implement Dijkstra's algorithm for a small network graph.
2. Simulate RIP (Distance Vector): each node updates from neighbors.
3. Model TCP AIMD window growth; visualize with Python.
4. Try RED: drop packets when queue length > threshold.
5. Compare TCP Reno, Cubic, BBR throughput in simulation.

86. Distributed Consensus (Paxos, Raft, PBFT)

In a distributed system, multiple nodes must agree on a single value, for example, the state of a log, a database entry, or a blockchain block. This agreement process is called consensus.

Consensus algorithms let distributed systems act as one reliable system, even when some nodes fail, crash, or lie (Byzantine faults).

1. Why Consensus?

Imagine a cluster managing a shared log (like in databases or Raft). Each node might:

- See different requests,- Fail and recover,- Communicate over unreliable links. We need all non-faulty nodes to agree on the same order of operations.

A valid consensus algorithm must satisfy:

- Agreement: all correct nodes choose the same value- Validity: the chosen value was proposed by a node- Termination: every correct node eventually decides- Fault Tolerance: works despite failures

2. The FLP Impossibility

The FLP theorem (Fischer, Lynch, Paterson, 1985) says:

> In an asynchronous system with even one faulty process, no deterministic algorithm can guarantee consensus.

So practical algorithms use:

- Randomization, or- Partial synchrony (timeouts, retries)

3. Paxos: The Classical Algorithm

Paxos, by Leslie Lamport, is the theoretical foundation for distributed consensus.

It revolves around three roles:

- Proposers: suggest values- Acceptors: vote on proposals- Learners: learn the final decision Consensus proceeds in two phases.

Phase 1 (Prepare)

1. Proposer picks a proposal number `n` and sends `(Prepare, n)` to acceptors.
2. Acceptors respond with their highest accepted proposal (if any).

Phase 2 (Accept)

1. If proposer receives a majority of responses, it sends `(Accept, n, v)` with value `v` (highest seen or new).
2. Acceptors accept if they haven't promised higher `n`.

When a majority accept, value `v` is chosen.

Guarantees

- Safety: no two different values chosen- Liveness: possible under stable leadership #### Drawbacks
- Complex to implement correctly- High messaging overhead > "Paxos is for theorists; Raft is for engineers."

4. Raft: Understandable Consensus

Raft was designed to be simpler and more practical than Paxos, focusing on replicated logs.

Roles

- Leader: coordinates all changes- Followers: replicate leader's log- Candidates: during elections #### Workflow

1. Leader Election

    - Timeout triggers candidate election. - Each follower votes; majority wins.2. Log Replication

    - Leader appends entries, sends `AppendEntries` RPCs. - Followers acknowledge; leader commits when majority ack.3. Safety

    - Logs are consistent across majority. - Followers accept only valid prefixes. Raft ensures:

- At most one leader per term- Committed entries never lost- Logs stay consistent #### Pseudocode Sketch

```
on timeout -> become_candidate()
send RequestVote(term, id)
if majority_votes -> become_leader()

on AppendEntries(term, entries):
    if term >= current_term:
        append(entries)
        reply success
```

5. PBFT: Byzantine Fault Tolerance

Paxos and Raft assume crash faults (nodes stop, not lie). For Byzantine faults (arbitrary behavior), we use PBFT (Practical Byzantine Fault Tolerance).

Tolerates up to `f` faulty nodes out of `3f + 1` total.

Phases

1. Pre-Prepare: Leader proposes value
2. Prepare: Nodes broadcast proposal hashes
3. Commit: Nodes confirm receipt by 2f+1 votes

Used in blockchains and critical systems (space, finance).

6. Quorum Concept

Consensus often relies on quorums (majorities):

- Two quorums always intersect, ensuring consistency.- Write quorum + read quorum total nodes. In Raft/Paxos:
- Majority = `N/2 + 1`- Guarantees overlap even if some nodes fail.

7. Log Replication and State Machines

Consensus underlies Replicated State Machines (RSM):

- Every node applies the same commands in the same order.- Guarantees deterministic, identical states. This model powers:
- Databases (etcd, Spanner, TiKV)- Coordination systems (ZooKeeper, Consul)- Kubernetes control planes

8. Leader Election

All practical consensus systems need leaders:

- Simplifies coordination- Reduces conflicts- Heartbeats detect failures- New elections restore progress Algorithms:
- Raft Election (random timeouts)- Bully Algorithm- Chang-Roberts Ring Election

9. Performance and Optimization

- Batching: amortize RPC overhead- Pipeline: parallelize appends- Read-only optimizations: serve from followers (stale reads)- Witness nodes: participate in quorum without full data Advanced:
- Multi-Paxos: reuse leader, fewer rounds- Fast Paxos: shortcut some phases- Viewstamped Replication: Paxos-like log replication

10. Why It Matters

Consensus is the backbone of reliability in modern distributed systems. Every consistent database, service registry, or blockchain depends on it.

Systems using consensus:

- etcd, Consul, ZooKeeper - cluster coordination- Raft in Kubernetes - leader election- PBFT in blockchains - fault-tolerant ledgers- Spanner, TiDB - consistent databases > "Consensus is how machines learn to agree, and trust."

Try It Yourself

1. Implement Raft leader election in C or Python.
2. Simulate Paxos on 5 nodes with message drops.
3. Explore PBFT: try failing nodes and Byzantine behavior.
4. Compare performance of Raft vs Paxos under load.
5. Build a replicated key-value store with Raft.

87. Load Balancing and Rate Limiting

When systems scale, no single server can handle all requests alone. Load balancing distributes incoming traffic across multiple servers to improve throughput, reduce latency, and prevent overload. Meanwhile, rate limiting protects systems by controlling how often requests are allowed, ensuring fairness, stability, and security.

These two ideas, spreading the load and controlling the flow, are cornerstones of modern distributed systems and APIs.

1. Why Load Balancing Matters

Imagine a web service receiving thousands of requests per second. If every request went to one machine, it would crash. A load balancer (LB) acts as a traffic director, spreading requests across many backends.

Goals:

- Efficiency - fully utilize servers- Reliability - no single point of failure- Scalability - handle growing workloads- Flexibility - add/remove servers dynamically

2. Types of Load Balancers

1. Layer 4 (Transport Layer)

Balances based on IP and port. Fast and protocol-agnostic (works for TCP/UDP).

Example: Linux IPVS, Envoy, HAProxy

2. Layer 7 (Application Layer)

Understands protocols like HTTP. Can route by URL path, headers, cookies.

Example: Nginx, Envoy, AWS ALB

3. Load Balancing Algorithms

Round Robin

Cycles through backends in order.

```
Req1 → ServerA
Req2 → ServerB
Req3 → ServerC
```

Simple, fair (if all nodes equal).

Weighted Round Robin

Assigns weights to reflect capacity. Example: ServerA(2x), ServerB(1x)

Least Connections

Send request to server with fewest active connections.

Least Response Time

Select backend with lowest latency (monitored dynamically).

Hash-Based (Consistent Hashing)

Deterministically route based on request key (like user ID).

- Keeps cache locality- Used in CDNs, distributed caches (e.g. memcached) #### Random

Pick a random backend, surprisingly effective under uniform load.

4. Consistent Hashing (In Depth)

Used for sharding and sticky sessions.

Key idea:

- Map servers to a hash ring- A request's key is hashed onto the ring- Assigned to next clockwise server When servers join/leave, only small fraction of keys move.

Used in:

- CDNs- Distributed caches (Redis Cluster, DynamoDB)- Load-aware systems

5. Health Checks and Failover

A smart LB monitors health of each server:

- Heartbeat pings (HTTP/TCP)- Auto-remove unhealthy servers- Rebalance traffic instantly Example: If ServerB fails, remove from rotation:

```
Healthy: [ServerA, ServerC]
```

Also supports active-passive failover: hot standby servers take over when active fails.

6. Global Load Balancing

Across regions or data centers:

- GeoDNS: route to nearest region- Anycast: advertise same IP globally; routing picks nearest- Latency-based routing: measure and pick lowest RTT Used by CDNs, cloud services, multi-region apps

7. Rate Limiting: The Other Side

If load balancing spreads the work, rate limiting keeps total work reasonable.

It prevents:

- Abuse (bots, DDoS)- Overload (too many requests)- Fairness issues (no user dominates resources) Policies:
- Per-user, per-IP, per-API-key- Global or per-endpoint

8. Rate Limiting Algorithms

Token Bucket

- Bucket holds tokens (capacity = burst limit)- Each request consumes 1 token- Tokens refill at constant rate (rate limit)- If empty → reject or delay Good for bursty traffic.

```
if (tokens > 0) {
    tokens--;
    allow();
} else reject();
```

Leaky Bucket

- Requests flow into a bucket, drain at fixed rate- Excess = overflow = dropped Smooths bursts; used for shaping.

Fixed Window Counter

- Count requests in fixed interval (e.g. 1s)- Reset every window- Simple but unfair around boundaries #### Sliding Window Log / Sliding Window Counter
- Maintain timestamps of requests- Remove old ones beyond time window- More accurate and fair

9. Combining Both

A full system might:

- Use rate limiting per user or service- Use load balancing across nodes- Apply circuit breakers when overload persists Together, they form resilient architectures that stay online even under spikes.

10. Why It Matters

These techniques make large-scale systems:

- Scalable - handle millions of users- Stable - prevent cascading failures- Fair - each client gets a fair share- Resilient - recover gracefully from spikes or node loss Used in:
- API Gateways (Kong, Envoy, Nginx)- Cloud Load Balancers (AWS ALB, GCP LB)- Kubernetes Ingress and Service Meshes- Distributed Caches and Databases > "Balance keeps systems alive. Limits keep them sane."

Try It Yourself

1. Simulate Round Robin and Least Connections balancing across 3 servers.
2. Implement a Token Bucket rate limiter in C or Python.
3. Test burst traffic, observe drops or delays.
4. Combine Consistent Hashing with Token Bucket for user-level control.
5. Visualize how load balancing + rate limiting keep system latency low.

88. Search and Indexing (Inverted, BM25, WAND)

Search engines, whether web-scale like Google or local like SQLite's FTS, rely on efficient indexing and ranking to answer queries fast. Instead of scanning all documents, they use indexes (structured lookup tables) to quickly find relevant matches.

This section explores inverted indexes, ranking algorithms (TF-IDF, BM25), and efficient retrieval techniques like WAND.

1. The Search Problem

Given:

- A corpus of documents- A query (e.g., “machine learning algorithms”) We want to return:
- Relevant documents- Ranked by importance and similarity Naive search → O(N × M) comparisons Inverted indexes → O(K log N), where K = terms in query

2. Inverted Index: The Heart of Search

An inverted index maps terms → documents containing them.

Example

Term	Postings List
“data”	[1, 4, 5]
“algorithm”	[2, 3, 5]
“machine”	[1, 2]

Each posting may include:

- docID- term frequency (tf)- positions (for phrase search) #### Construction Steps

1. Tokenize documents → words
2. Normalize (lowercase, stemming, stopword removal)
3. Build postings: term → [docIDs, tf, positions]
4. Sort & compress for storage efficiency

Used by:

- Elasticsearch, Lucene, Whoosh, Solr

3. Boolean Retrieval

Simplest model:

- Query = Boolean expression e.g. `(machine AND learning) OR AI`

Use set operations on postings:

- AND $\rightarrow$ intersection- OR $\rightarrow$ union- NOT $\rightarrow$ difference Fast intersection uses merge algorithm on sorted lists.

```
void intersect(int A[], int B[], int n, int m) {
    int i = 0, j = 0;
    while (i < n && j < m) {
        if (A[i] == B[j]) { print(A[i]); i++; j++; }
        else if (A[i] < B[j]) i++;
        else j++;
    }
}
```

But Boolean search doesn't rank results, so we need scoring models.

4. Vector Space Model

Represent documents and queries as term vectors. Each dimension = term weight (tf-idf).

- tf: term frequency in document- idf: inverse document frequency $idf = \log \frac{N}{df_t}$

Cosine similarity measures relevance:

$$\text{score}(q, d) = \frac{q \cdot d}{|q||d|}$$

Simple, interpretable, forms basis of BM25 and modern embeddings.

5. BM25: The Classic Ranking Function

BM25 (Best Match 25) is the de facto standard in information retrieval.

$$\text{score}(q,d) = \sum_{t \in q} IDF(t) \cdot \frac{f(t,d) \cdot (k_1 + 1)}{f(t,d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{avgdl})}$$

Where:

- (f(t, d)): term frequency- (|d|): doc length- (avgdl): average doc length- k_1, b: tunable params (typ. 1.2-2.0, 0.75) #### Advantages
- Balances term frequency, document length, and rarity- Fast and effective baseline- Still used in Elasticsearch, Lucene, OpenSearch

6. Efficiency Tricks: WAND, Block-Max WAND

Ranking involves merging multiple postings. We can skip irrelevant documents early with WAND (Weak AND).

WAND Principle

- Each term has upper-bound score- Maintain pointers in each posting- Compute potential max score- If max < current threshold, skip doc Improves latency for top-k retrieval.

Variants:

- BMW (Block-Max WAND) - uses block-level score bounds- MaxScore - simpler thresholding- Dynamic pruning - skip unpromising candidates

7. Index Compression

Postings lists are long, compression is crucial.

Common schemes:

- Delta encoding: store gaps between docIDs- Variable-byte (VB) or Gamma coding- Frame of Reference (FOR) and SIMD-BP128 for vectorized decoding Goal: smaller storage + faster decompression

8. Advanced Retrieval

Proximity Search

Require words appear near each other. Use positional indexes.

Phrase Search

Match exact sequences using positions: “machine learning”   “learning machine”

Fuzzy / Approximate Search

Allow typos: Use Levenshtein automata, n-grams, or k-approximate matching

Fielded Search

Score per field (title, body, tags) Weighted combination

9. Learning-to-Rank and Semantic Search

Modern search adds ML-based re-ranking:

- Learning to Rank (LTR): use features (tf, idf, BM25, clicks)- Neural re-ranking: BERT-style embeddings for semantic similarity- Hybrid retrieval: combine BM25 + dense vectors (e.g. ColBERT, RRF) Also: ANN (Approximate Nearest Neighbor) for vector-based search.

10. Why It Matters

Efficient search powers:

- Web search engines- IDE symbol lookup- Log search, code search- Database full-text search- AI retrieval pipelines (RAG) It’s where algorithms meet language and scale.

  “Search is how we connect meaning to memory.”

Try It Yourself

1. Build a tiny inverted index in C or Python.
2. Implement Boolean AND and OR queries.
3. Compute TF-IDF and BM25 scores for a toy dataset.
4. Add WAND pruning for top-k retrieval.
5. Compare BM25 vs semantic embeddings for relevance.

89. Compression and Encoding in Systems

Compression and encoding algorithms are the quiet workhorses of computing, shrinking data to save space, bandwidth, and time. They allow systems to store more, transmit faster, and process efficiently. From files and databases to networks and logs, compression shapes nearly every layer of system design.

1. Why Compression Matters

Compression is everywhere:

- Databases - column stores, indexes, logs- Networks - HTTP, TCP, QUIC payloads- File systems - ZFS, NTFS, btrfs compression- Streaming - video/audio codecs- Logs & telemetry - reduce I/O and storage cost Benefits:
- Smaller data = faster I/O- Less storage = lower cost- Less transfer = higher throughput Trade-offs:
- CPU overhead (compression/decompression)- Latency (especially for small data)- Suitability (depends on entropy and structure)

2. Key Concepts

Entropy

Minimum bits needed to represent data (Shannon). High entropy $\rightarrow$ less compressible.

Redundancy

Compression exploits repetition and patterns.

Lossless vs Lossy

- Lossless: reversible (ZIP, PNG, LZ4)- Lossy: approximate (JPEG, MP3, H.264) In system contexts, lossless dominates.

3. Common Lossless Compression Families

Huffman Coding

- Prefix-free variable-length codes- Frequent symbols = short codes- Optimal under symbol-level model Used in: DEFLATE, JPEG, MP3

Arithmetic Coding

- Encodes sequence as fractional interval- More efficient than Huffman for skewed distributions- Used in: H.264, bzip2, AV1 #### Dictionary-Based (LZ77, LZ78)
- Replace repeated substrings with references- Core of ZIP, gzip, zlib, LZMA, Snappy #### LZ77 Sketch

```
while (not EOF) {
    find longest match in sliding window;
    output (offset, length, next_char);
}
```

Variants:

- LZ4 - fast, lower ratio- Snappy - optimized for speed- Zstandard (Zstd) - tunable speed/ratio, dictionary support #### Burrows-Wheeler Transform (BWT)
- Reorders data to group similar symbols- Followed by Move-To-Front + Huffman- Used in bzip2, BWT-based compressors #### Run-Length Encoding (RLE)
- Replace consecutive repeats with (symbol, count)- Great for structured or sparse data Example: `AAAAABBBCC` → `(A,5)(B,3)(C,2)`

4. Specialized Compression in Systems

Columnar Databases

Compress per column:

- Dictionary encoding - map strings → ints- Run-length encoding - good for sorted columns- Delta encoding - store differences (time series)- Bit-packing - fixed-width integers in minimal bits Combine multiple for optimal ratio.

Example (time deltas):

```
[100, 102, 103, 107] → [100, +2, +1, +4]
```

Log and Telemetry Compression

- Structured formats → fieldwise encoding- Often Snappy or LZ4 for fast decode- Aggregators (Fluentd, Loki, Kafka) rely heavily on them #### Data Lakes and Files
- Parquet, ORC, Arrow → columnar + compressed- Choose codec per column: LZ4 for speed, Zstd for ratio

5. Streaming and Chunked Compression

Large data often processed in chunks:

- Enables random access and parallelism- Needed for network streams, distributed files Example: `zlib` block, `Zstd` frame, `gzip` chunk

Used in:

- HTTP chunked encoding- Kafka log segments- MapReduce shuffle

6. Encoding Schemes

Compression encoding. Encoding ensures safe transport.

Base64

- Maps 3 bytes → 4 chars- 33% overhead- Used for binary → text (emails, JSON APIs) #### URL Encoding
- Escape unsafe chars with `%xx` #### Delta Encoding
- Store differences, not full values #### Varint / Zigzag Encoding
- Compact integers (e.g. protobufs)- Smaller numbers → fewer bytes Example:

```
while (x >= 0x80) { emit((x & 0x7F) | 0x80); x >>= 7; }
emit(x);
```

7. Adaptive and Context Models

Modern compressors adapt to local patterns:

- PPM (Prediction by Partial Matching)- Context mixing (PAQ)- Zstd uses FSE (Finite State Entropy) coding Balance between speed, memory, and compression ratio.

8. Hardware Acceleration

Compression can be offloaded to:

- CPUs with SIMD (AVX2, SSE4.2)- GPUs (parallel encode/decode)- NICs / SmartNICs- ASICs (e.g., Intel QAT) Critical for high-throughput databases, network appliances, storage systems.

9. Design Trade-offs

Goal	Best Choice
Max speed	LZ4, Snappy
Max ratio	Zstd, LZMA
Balance	Zstd (tunable)
Column store	RLE, Delta, Dict
Logs / telemetry	Snappy, LZ4
Archival	bzip2, xz
Real-time	LZ4, Brotli (fast mode)

Choose based on CPU budget, I/O cost, latency tolerance.

10. Why It Matters

Compression is a first-class optimization:

- Saves petabytes in data centers- Boosts throughput across networks- Powers cloud storage (S3, BigQuery, Snowflake)- Enables efficient analytics and ML pipelines > "Every byte saved is time earned."

Try It Yourself

1. Compress text using Huffman coding (build frequency table).
2. Compare gzip, Snappy, and Zstd on a 1GB dataset.
3. Implement delta encoding and RLE for numeric data.
4. Try dictionary encoding on repetitive strings.
5. Measure compression ratio, speed, and CPU usage trade-offs.

90. Fault Tolerance and Replication

Modern systems must survive hardware crashes, network partitions, or data loss without stopping. Fault tolerance ensures that a system continues to function, even when parts fail. Replication underpins this resilience, duplicating data or computation across multiple nodes for redundancy, performance, and consistency.

Together, they form the backbone of reliability in distributed systems.

1. Why Fault Tolerance?

No system is perfect:

- Servers crash- Disks fail- Networks partition- Power goes out The question isn't *if* failure happens, but *when.* Fault-tolerant systems detect, contain, and recover from failure automatically.

Goals:

- Availability - keep serving requests- Durability - never lose data- Consistency - stay correct across replicas

2. Failure Models

Crash Faults

Node stops responding but doesn't misbehave. Handled by restarts or replication (Raft, Paxos).

Omission Faults

Lost messages or dropped updates. Handled with retries and acknowledgments.

Byzantine Faults

Arbitrary/malicious behavior. Handled by Byzantine Fault Tolerance (PBFT), expensive but robust.

3. Redundancy: The Core Strategy

Fault tolerance = redundancy + detection + recovery

Redundancy types:

- Hardware: multiple power supplies, disks (RAID)- Software: replicated services, retries- Data: multiple copies, erasure codes- Temporal: retry or checkpoint and replay

4. Replication Models

1. Active Replication

All replicas process requests in parallel (lockstep). Results must match. Used in real-time and Byzantine systems.

2. Passive (Primary-Backup)

One leader (primary) handles requests. Backups replicate log, take over on failure. Used in Raft, ZooKeeper, PostgreSQL streaming.

3. Quorum Replication

Writes and reads contact majority of replicas. Ensures overlap → consistency. Used in Cassandra, DynamoDB, Etcd.

5. Consistency Models

Replication introduces a trade-off between consistency and availability (CAP theorem).

Strong Consistency

All clients see the same value immediately. Example: Raft, Etcd, Spanner.

Eventual Consistency

Replicas converge over time. Example: DynamoDB, Cassandra.

Causal Consistency

Preserves causal order of events. Example: Vector clocks, CRDTs.

Choice depends on workload:

- Banking → strong- Social feeds → eventual- Collaborative editing → causal

6. Checkpointing and Recovery

To recover after crash:

- Periodically checkpoint state- On restart, replay log of missed events Example: Databases → Write-Ahead Log (WAL) Stream systems → Kafka checkpoints

```
1. Save state to disk
2. Record latest log position
3. On restart → reload + replay
```

7. Erasure Coding

Instead of full copies, store encoded fragments. With (k) data blocks, (m) parity blocks → tolerate (m) failures.

Example: Reed-Solomon (used in HDFS, Ceph)

k	m	Total	Fault Tolerance
4	2	6	2 failures

Better storage efficiency than 3× replication.

8. Failure Detection

Detecting failure is tricky in distributed systems (because of latency). Common techniques:

- Heartbeats - periodic "I'm alive" messages- Timeouts - suspect node if no heartbeat-Gossip protocols - share failure info among peers Used in Consul, Cassandra, Kubernetes health checks.

9. Self-Healing Systems

After failure:

1. Detect it
2. Isolate faulty component
3. Replace or restart
4. Rebalance load or re-replicate data

Patterns:

- Supervisor trees (Erlang/Elixir)- Self-healing clusters (Kubernetes)- Rebalancing (Cassandra ring repair) "Never trust a single machine, trust the system."

10. Why It Matters

Fault tolerance turns fragile infrastructure into reliable services.

Used in:

- Databases (replication + WAL)- Distributed storage (HDFS, Ceph, S3)- Orchestration (Kubernetes controllers)- Streaming systems (Kafka, Flink) Without replication and fault tolerance, large-scale systems would collapse under failure.

  "Resilience is built, not assumed."

Try It Yourself

1. Build a primary-backup key-value store: leader writes, follower replicates.
2. Add heartbeat + timeout detection to trigger failover.
3. Simulate partition: explore behavior under strong vs eventual consistency.
4. Implement checkpoint + replay recovery for a small app.
5. Compare 3× replication vs Reed-Solomon (4+2) for space and reliability.

Chapter 10. AI, ML and Optimization

91. Classical ML (k-means, Naive Bayes, SVM, Decision Trees)

Classical machine learning is built on interpretable mathematics and solid optimization foundations. Long before deep learning, these algorithms powered search engines, spam filters, and recommendation systems. They're still used today, fast, explainable, and easy to deploy.

This section covers the four pillars of classical ML:

- k-means - unsupervised clustering- Naive Bayes - probabilistic classification- SVM - margin-based classification- Decision Trees - rule-based learning

1. The Essence of Classical ML

Classical ML is about learning from data using statistical principles, often without huge compute. Given dataset (D = $\{x_i, y_i\}$), the task is to:

- Predict (y) from (x)- Generalize beyond seen data- Balance bias and variance

2. k-means Clustering

Goal: partition data into (k) groups (clusters) such that intra-cluster distance is minimized.

Objective

$$\min_{C} \sum_{i=1}^{k} \sum_{x \in C_i} |x - \mu_i|^2$$

Where μ_i = centroid of cluster (i).

Algorithm

1. Choose (k) random centroids
2. Assign each point to nearest centroid
3. Recompute centroids
4. Repeat until stable

Tiny Code (C-style)

```
for (iter = 0; iter < max_iter; iter++) {
    assign_points_to_clusters();
    recompute_centroids();
}
```

Pros

- Simple, fast ((O(nkd)))- Works well for spherical clusters #### Cons
- Requires (k)- Sensitive to initialization, outliers Variants:
- k-means++ (better initialization)- Mini-batch k-means (scalable)

3. Naive Bayes Classifier

A probabilistic model using Bayes' theorem under independence assumptions.

$$P(y|x) \propto P(y) \prod_{i=1}^{n} P(x_i|y)$$

Algorithm

1. Compute prior (P(y))
2. Compute likelihood (P$x_i|y$)
3. Predict class with max posterior

Types

- Multinomial NB - text (bag of words)- Gaussian NB - continuous features- Bernoulli NB - binary features #### Example (Spam Detection)

```
P(spam | "win money")  P(spam) * P("win"|spam) * P("money"|spam)
```

Pros

- Fast, works well for text- Needs little data- Probabilistic interpretation #### Cons
- Assumes feature independence- Poor for correlated features

4. Support Vector Machines (SVM)

SVM finds the max-margin hyperplane separating classes.

Objective

Maximize margin = distance between boundary and nearest points.

$$\min_{w,b} \frac{1}{2}|w|^2 \quad \text{s.t.} \quad y_i(w \cdot x_i + b) \geq 1$$

Can be solved via Quadratic Programming.

Intuition

- Each data point → vector- Hyperplane: $w \cdot x + b = 0$- Support vectors = boundary points #### Kernel Trick

Transform input via kernel (K$x_i, x_j = \ x_i \quad x_j$):

- Linear: dot product- Polynomial: ($x_i \cdot x_j + c$^d)- RBF: $e^{-\gamma|x_i - x_j|^2}$ #### Pros
- Effective in high dimensions- Can model nonlinear boundaries- Few hyperparameters #### Cons
- Slow on large data- Harder to tune kernel parameters

5. Decision Trees

If-else structure for classification/regression.

At each node:

- Pick feature (f) and threshold (t)- Split to maximize information gain #### Metrics
- Entropy: $H = -\sum p_i \log p_i$- Gini: $G = 1 - \sum p_i^2$ #### Pseudocode

```
if (feature < threshold)
    go left;
else
    go right;
```

Build recursively until:

- Max depth- Min samples per leaf- Pure nodes #### Pros
- Interpretable- Handles mixed data- No scaling needed #### Cons
- Prone to overfitting- Unstable (small data changes) Fixes:
- Pruning (reduce depth)- Ensembles: Random Forests, Gradient Boosting

Algorithm	Bias	Variance

6. Bias-Variance Tradeoff

Algorithm	Bias	Variance
k-means	High	Low
Naive Bayes	High	Low
SVM	Low	Medium
Decision Tree	Low	High

Balancing both = good generalization.

7. Evaluation Metrics

For classification:

- Accuracy, Precision, Recall, F1-score- ROC-AUC, Confusion Matrix For clustering:
- Inertia, Silhouette Score Always use train/test split or cross-validation.

8. Scaling to Large Data

Techniques:

- Mini-batch training- Online updates (SGD)- Dimensionality reduction (PCA)- Approximation (Random Projections) Libraries:
- scikit-learn (Python)- liblinear, libsvm (C/C++)- MLlib (Spark)

9. When to Use What

Task	Recommended Algorithm
Text classification	Naive Bayes
Clustering	k-means
Nonlinear classification	SVM (RBF)
Tabular data	Decision Tree
Quick baseline	Logistic Regression / NB

10. Why It Matters

These algorithms are fast, interpretable, and theoretical foundations of modern ML. They remain the go-to choice for:

- Small to medium datasets- Real-time classification- Explainable AI > "Classical ML is the art of solving problems with math you can still write on a whiteboard."

Try It Yourself

1. Cluster 2D points with k-means, plot centroids.
2. Train Naive Bayes on a spam/ham dataset.
3. Classify linearly separable data with SVM.
4. Build a Decision Tree from scratch (entropy, Gini).
5. Compare models' accuracy and interpretability.

92. Ensemble Methods (Bagging, Boosting, Random Forests)

Ensemble methods combine multiple weak learners to build a strong predictor. Instead of relying on one model, ensembles vote, average, or boost multiple models, improving stability and accuracy.

They are the bridge between classical and modern ML , simple models, combined smartly, become powerful.

1. The Core Idea

> "Many weak learners, when combined, can outperform a single strong one."

Mathematically, if $f_1, f_2, \ldots, f_k$ are weak learners, an ensemble predictor is:

$$F(x) = \frac{1}{k} \sum_{i=1}^{k} f_i(x)$$

For classification, combine via majority vote. For regression, combine via average.

2. Bagging (Bootstrap Aggregating)

Bagging reduces variance by training models on different samples.

Steps

1. Draw (B) bootstrap samples from dataset (D).
2. Train one model per sample.
3. Aggregate predictions by averaging or voting.

$$\hat{f} * bag(x) = \frac{1}{B} \sum *b = 1^B f_b(x)$$

Each f_b is trained on a random subset (with replacement).

Example

- Base learner: Decision Tree- Ensemble: Bagged Trees- Famous instance: Random Forest #### Tiny Code (C-style Pseudocode)

```
for (int b = 0; b < B; b++) {
    D_b = bootstrap_sample(D);
    model[b] = train_tree(D_b);
}
prediction = average_predictions(model, x);
```

Pros

- Reduces variance- Works well with high-variance learners- Parallelizable #### Cons
- Increases computation- Doesn't reduce bias

3. Random Forest

A bagging-based ensemble of decision trees with feature randomness.

Key Ideas

- Each tree trained on bootstrap sample.- At each split, consider random subset of features.- Final prediction = majority vote or average. This decorrelates trees, improving generalization.

$$F(x) = \frac{1}{B} \sum_{b=1}^{B} T_b(x)$$

Pros

- Handles large feature sets- Low overfitting- Good default for tabular data #### Cons
- Less interpretable- Slower on huge datasets OOB (Out-of-Bag) error = internal validation from unused samples.

4. Boosting

Boosting focuses on reducing bias by sequentially training models , each one corrects errors from the previous.

Steps

1. Start with weak learner (f_1(x))
2. Train next learner (f_2(x)) on residuals/errors
3. Combine with weighted sum

$$F_m(x) = F_{m-1}(x) + \alpha_m f_m(x)$$

Weights α_m focus on difficult examples.

Tiny Code (Conceptual)

```
F = 0;
for (int m = 0; m < M; m++) {
    residual = y - predict(F, x);
    f_m = train_weak_learner(x, residual);
    F += alpha[m] * f_m;
}
```

5. AdaBoost (Adaptive Boosting)

AdaBoost adapts weights on samples after each iteration.

Algorithm

1. Initialize weights: $w_i = \frac{1}{n}$
2. Train weak classifier f_t
3. Compute error: ϵ_t
4. Update weights:
$$w_i \leftarrow w_i \cdot e^{\alpha_t \cdot I(y_i \neq f_t(x_i))}$$
where $\alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$
5. Normalize weights

Final classifier:
$$F(x) = \text{sign} \left(\sum_t \alpha_t f_t(x) \right)$$

Pros

- High accuracy on clean data- Simple and interpretable weights #### Cons
- Sensitive to outliers- Sequential $\rightarrow$ not easily parallelizable

6. Gradient Boosting

A modern version of boosting using gradient descent on loss.

At each step, fit new model to negative gradient of loss function.

Objective

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

where $h_m(x) \approx -\frac{\partial L(y, F(x))}{\partial F(x)}$

Common Libraries

- XGBoost
- LightGBM
- CatBoost #### Pros
- High performance on tabular data- Flexible (custom loss)- Handles mixed feature types #### Cons
- Slower to train- Sensitive to hyperparameters

7. Stacking (Stacked Generalization)

Combine multiple models (base learners) via a meta-model.

Steps

1. Train base models (SVM, Tree, NB, etc.)
2. Collect their predictions
3. Train meta-model (e.g. Logistic Regression) on outputs

$$\hat{y} = f_{meta}(f_1(x), f_2(x), \dots, f_k(x))$$

8. Bagging vs Boosting

Feature	Bagging	Boosting
Strategy	Parallel	Sequential
Goal	Reduce variance	Reduce bias
Weighting	Uniform	Adaptive
Example	Random Forest	AdaBoost, XGBoost

9. Bias-Variance Behavior

- Bagging: ↓ variance- Boosting: ↓ bias- Random Forest: balanced- Stacking: flexible but complex

10. Why It Matters

Ensemble methods are the workhorses of classical ML competitions and real-world tabular problems. They blend interpretability, flexibility, and predictive power.

> "One tree may fall, but a forest stands strong."

Try It Yourself

1. Train a Random Forest on the Iris dataset.
2. Implement AdaBoost from scratch using decision stumps.
3. Compare Bagging vs Boosting accuracy.
4. Try XGBoost with different learning rates.
5. Visualize feature importance across models.

93. Gradient Methods (SGD, Adam, RMSProp)

Gradient-based optimization is the heartbeat of machine learning. These methods update parameters iteratively by following the negative gradient of the loss function. They power everything from linear regression to deep neural networks.

1. The Core Idea

We want to minimize a loss function (Lθ). Starting from some initial parameters θ_0, we move in the opposite direction of the gradient:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta L(\theta_t)$$

where η is the learning rate (step size).

The gradient tells us which way the function increases fastest , we move the other way.

2. Batch Gradient Descent

Uses the entire dataset to compute the gradient.

$$\nabla_\theta L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \ell_i(\theta)$$

- Accurate but slow for large (N)- Each update is expensive Tiny Code

```
for (int t = 0; t < T; t++) {
    grad = compute_full_gradient(data, theta);
    theta = theta - eta * grad;
}
```

Good for: small datasets or convex problems

3. Stochastic Gradient Descent (SGD)

Instead of full data, use one random sample per step.

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta \ell_i(\theta_t)$$

- Noisy but faster updates- Can escape local minima- Great for online learning Tiny Code

```
for each sample (x_i, y_i):
    grad = grad_loss(theta, x_i, y_i);
    theta -= eta * grad;
```

Pros

- Fast convergence- Works on large datasets Cons
- Noisy updates- Requires learning rate tuning

4. Mini-Batch Gradient Descent

Compromise between batch and stochastic.

Use small subset (mini-batch) of samples:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \ell_i(\theta_t)$$

Usually batch size = 32 or 64. Faster, more stable updates.

5. Momentum

Adds velocity to smooth oscillations.

$$v_t = \beta v_{t-1} + (1-\beta)\nabla_\theta L(\theta_t)$$

$$\theta_{t+1} = \theta_t - \eta v_t$$

This accumulates past gradients to speed movement in consistent directions.

Think of it like a heavy ball rolling down a hill.

6. Nesterov Accelerated Gradient (NAG)

Improves momentum by looking ahead:

$$v_t = \beta v_{t-1} + \eta \nabla_\theta L(\theta_t - \beta v_{t-1})$$

It anticipates the future position before computing the gradient.

Faster convergence in convex settings.

7. RMSProp

Adjusts learning rate per parameter using exponential average of squared gradients:

$$E[g^2] * t = \rho E[g^2] * t - 1 + (1-\rho) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

This helps when gradients vary in magnitude.

Good for: non-stationary objectives, deep networks

8. Adam (Adaptive Moment Estimation)

Combines momentum + RMSProp:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Bias-corrected estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Tiny Code (Conceptual)

```
m = 0; v = 0;
for (int t = 1; t <= T; t++) {
    g = grad(theta);
    m = beta1 * m + (1 - beta1) * g;
    v = beta2 * v + (1 - beta2) * g * g;
    m_hat = m / (1 - pow(beta1, t));
    v_hat = v / (1 - pow(beta2, t));
    theta -= eta * m_hat / (sqrt(v_hat) + eps);
}
```

Pros

- Works well out of the box- Adapts learning rate- Great for deep learning Cons
- May not converge exactly- Needs decay schedule for stability

9. Learning Rate Schedules

Control η over time:

- Step decay: $\eta_t = \eta_0 \cdot \gamma^{\lfloor t/s \rfloor}$- Exponential decay: $\eta_t = \eta_0 e^{-\lambda t}$- Cosine annealing: smooth periodic decay- Warm restarts: reset learning rate periodically

10. Why It Matters

All modern deep learning is built on gradients. Choosing the right optimizer can mean faster training and better accuracy.

Optimizer	Adaptive	Momentum	Common Use
SGD	No	Optional	Simple tasks
SGD + Momentum	No	Yes	ConvNets
RMSProp	Yes	No	RNNs
Adam	Yes	Yes	Transformers, DNNs

> "Optimization is the art of taking small steps in the right direction , many times over."

Try It Yourself

1. Implement SGD and Adam on a linear regression task.
2. Compare training curves for SGD, Momentum, RMSProp, and Adam.
3. Experiment with learning rate schedules.
4. Visualize optimization paths on a 2D contour plot.

94. Deep Learning (Backpropagation, Dropout, Normalization)

Deep learning is about stacking layers of computation so that the network can learn hierarchical representations. From raw pixels to abstract features, deep nets build meaning through composition of functions.

At the core of this process are three ideas: backpropagation, regularization (dropout), and normalization.

1. The Essence of Deep Learning

A neural network is a chain of functions:

$$f(x;\theta) = f_L(f_{L-1}(\cdots f_1(x)))$$

Each layer transforms its input and passes it on.

Training involves finding parameters θ that minimize a loss (L(fx; θ, y)).

2. Backpropagation

Backpropagation is the algorithm that teaches neural networks.

It uses the chain rule of calculus to efficiently compute gradients layer by layer.

For each layer (i):

$$\frac{\partial L}{\partial \theta_i} = \frac{\partial L}{\partial a_i} \cdot \frac{\partial a_i}{\partial \theta_i}$$

and propagate backward:

$$\frac{\partial L}{\partial a_{i-1}} = \frac{\partial L}{\partial a_i} \cdot \frac{\partial a_i}{\partial a_{i-1}}$$

So every neuron learns how much it contributed to the error.

Tiny Code

```
// Pseudocode for 2-layer network
forward:
    z1 = W1*x + b1;
    a1 = relu(z1);
    z2 = W2*a1 + b2;
    y_hat = softmax(z2);
    loss = cross_entropy(y_hat, y);

backward:
    dz2 = y_hat - y;
    dW2 = dz2 * a1.T;
    db2 = sum(dz2);
    da1 = W2.T * dz2;
    dz1 = da1 * relu_grad(z1);
    dW1 = dz1 * x.T;
    db1 = sum(dz1);
```

Each gradient is computed by local differentiation and multiplied back.

3. Activation Functions

Nonlinear activations let networks approximate nonlinear functions.

Function	Formula	Use
ReLU	$\max(0, x)$	Default, fast
Sigmoid	$\frac{1}{1+e^{-x}}$	Probabilities
Tanh	$\tanh(x)$	Centered activations
GELU	$x\,\Phi(x)$	Modern transformers

Without nonlinearity, stacking layers is just one big linear transformation.

4. Dropout

Dropout is a regularization technique that prevents overfitting. During training, randomly turn off neurons:

$$\tilde{a}_i = a_i \cdot m_i, \quad m_i \sim \text{Bernoulli}(p)$$

At inference, scale activations by (p) (keep probability).

It forces the network to not rely on any single path.

Tiny Code

```
for (int i = 0; i < n; i++) {
    if (rand_uniform() < p) a[i] = 0;
    else a[i] /= p; // scaling
}
```

5. Normalization

Normalization stabilizes and speeds up training by reducing internal covariate shift.

Batch Normalization

Normalize activations per batch:

$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y = \gamma\hat{x} + \beta$$

Learnable parameters γ, β restore flexibility.

Benefits:

- Smooth gradients- Allows higher learning rates- Acts as regularizer #### Layer Normalization

Used in transformers (normalizes across features, not batch).

6. Initialization

Proper initialization helps gradients flow.

Scheme	Formula	Use
Xavier	$\mathrm{Var}(W) = \frac{1}{n_{in}}$	Tanh
He	$\mathrm{Var}(W) = \frac{2}{n_{in}}$	ReLU

Poor initialization can lead to vanishing or exploding gradients.

7. Training Pipeline

1. Initialize weights
2. Forward pass
3. Compute loss
4. Backward pass (backprop)
5. Update weights (e.g. with Adam)

Repeat until convergence.

8. Deep Architectures

Model	Key Idea	Typical Use
MLP	Fully connected	Tabular data
CNN	Convolutions	Images
RNN	Sequential recurrence	Time series, text
Transformer	Self-attention	Language, vision

Each architecture stacks linear operations and nonlinearities in different ways.

9. Overfitting and Regularization

Common fixes:

- Dropout- Weight decay (L_2 regularization)- Data augmentation- Early stopping The key is to improve generalization, not just minimize training loss.

10. Why It Matters

Backpropagation turned neural networks from theory to practice. Normalization made them train faster. Dropout made them generalize better.

Together, they unlocked the deep learning revolution.

> "Depth gives power, but gradients give life."

Try It Yourself

1. Implement a 2-layer network with ReLU and softmax.
2. Add dropout and batch normalization.
3. Visualize training with and without dropout.
4. Compare performance on MNIST with and without normalization.

95. Sequence Models (Viterbi, Beam Search, CTC)

Sequence models process data where order matters, text, speech, DNA, time series. They capture dependencies across positions, predicting the next step from context.

This section explores three fundamental tools: Viterbi, Beam Search, and CTC (Connectionist Temporal Classification).

1. The Nature of Sequential Data

Sequential data has temporal or structural order. Each element x_t depends on past inputs $x_{1:t-1}$.

Common sequence tasks:

- Tagging (POS tagging, named entity recognition)- Transcription (speech $\rightarrow$ text)- Decoding (translation, path reconstruction) To handle such problems, we need models that remember.

2. Hidden Markov Models (HMMs)

A Hidden Markov Model assumes:

- A sequence of hidden states $z_1, z_2, \dots, z_T$. Each state emits an observation x_t. Transition and emission probabilities govern the process

$$P(z_t|z_{t-1}) = A_{z_{t-1},z_t}, \quad P(x_t|z_t) = B_{z_t}(x_t)$$

Goal: find the most likely sequence of hidden states given observations.

3. The Viterbi Algorithm

Viterbi is a dynamic programming algorithm to decode the most probable path:

$$\delta_t(i) = \max_{z_{1:t-1}} P(z_{1:t-1}, z_t = i, x_{1:t})$$

Recurrence:

$$\delta_t(i) = \max_j \left(\delta_{t-1}(j) \cdot A_{j,i}\right) \cdot B_i(x_t)$$

Track backpointers to reconstruct the best sequence.

Time complexity: $O(T \cdot N^2)$,
where $N =$ number of states, $T =$ sequence length.

Tiny Code

```
for (t = 1; t < T; t++) {
    for (i = 0; i < N; i++) {
        double best = -INF;
        int argmax = -1;
        for (j = 0; j < N; j++) {
            double score = delta[t-1][j] * A[j][i];
            if (score > best) { best = score; argmax = j; }
        }
        delta[t][i] = best * B[i][x[t]];
        backptr[t][i] = argmax;
    }
}
```

Use `backptr` to trace back the optimal path.

4. Beam Search

For many sequence models (e.g. neural machine translation), exhaustive search is impossible. Beam search keeps only the top-k best hypotheses at each step.

Algorithm:

1. Start with an empty sequence and score 0
2. At each step, expand each candidate with all possible next tokens
3. Keep only k best sequences (beam size)
4. Stop when all sequences end or reach max length

Beam size controls trade-off:

- Larger beam → better accuracy, slower- Smaller beam → faster, riskier

Tiny Code

```
for (step = 0; step < max_len; step++) {
    vector<Candidate> new_beam;
    for (c in beam) {
        probs = model_next(c.seq);
        for (token, p in probs)
            new_beam.push({c.seq + token, c.score + log(p)});
    }
    beam = top_k(new_beam, k);
}
```

Use log probabilities to avoid underflow.

5. Connectionist Temporal Classification (CTC)

Used in speech recognition and handwriting recognition where input and output lengths differ.

CTC learns to align input frames with output symbols without explicit alignment.

Add a special blank symbol ( ) to allow flexible alignment.

Example (CTC decoding):

Frame	Output	After Collapse
A A A	A A	A A
H H	H H	H

Loss:

$$P(y|x) = \sum_{\pi \in \text{Align}(x,y)} P(\pi|x)$$

where π are all alignments that reduce to (y).

CTC uses dynamic programming to compute forward-backward probabilities.

6. Comparing Methods

Method	Used In	Key Idea	Handles Alignment?
Viterbi	HMMs	Most probable state path	Yes
Beam Search	Neural decoders	Approximate search	Implicit
CTC	Speech / seq2seq	Sum over alignments	Yes

7. Use Cases

- Viterbi: POS tagging, speech decoding- Beam Search: translation, text generation- CTC: ASR, OCR, gesture recognition

8. Implementation Tips

- Use log-space for probabilities- In beam search, apply length normalization- In CTC, use dynamic programming tables- Combine CTC + beam search for speech decoding

9. Common Pitfalls

- Viterbi assumes Markov property (limited memory)- Beam Search can miss global optimum- CTC can confuse repeated characters without blanks

10. Why It Matters

Sequence models are the bridge between structure and time. They show how to decode hidden meaning in ordered data.

From decoding Morse code to transcribing speech, these algorithms give machines the gift of sequence understanding.

Try It Yourself

1. Implement Viterbi for a 3-state HMM.
2. Compare greedy decoding vs beam search on a toy language model.
3. Build a CTC loss table for a short sequence (like "HELLO").

96. Metaheuristics (GA, SA, PSO, ACO)

Metaheuristics are general-purpose optimization strategies that search through vast, complex spaces when exact methods are too slow or infeasible. They don't guarantee the perfect answer but often find good-enough solutions fast.

This section covers four classics:

- GA (Genetic Algorithm)- SA (Simulated Annealing)- PSO (Particle Swarm Optimization)- ACO (Ant Colony Optimization)

1. The Metaheuristic Philosophy

Metaheuristics draw inspiration from nature and physics. They combine exploration (searching widely) and exploitation (refining promising spots).

They're ideal for:

- NP-hard problems (TSP, scheduling)- Continuous optimization (parameter tuning)- Black-box functions (no gradients) They trade mathematical guarantees for practical power.

2. Genetic Algorithm (GA)

Inspired by natural selection, GAs evolve a population of solutions.

Core Steps

1. Initialize population randomly
2. Evaluate fitness of each
3. Select parents
4. Crossover to produce offspring
5. Mutate to add variation
6. Replace worst with new candidates

Repeat until convergence.

Tiny Code

```
for (gen = 0; gen < max_gen; gen++) {
    evaluate(pop);
    parents = select_best(pop);
    offspring = crossover(parents);
    mutate(offspring);
    pop = select_survivors(pop, offspring);
}
```

Operators

- Selection: tournament, roulette-wheel- Crossover: one-point, uniform- Mutation: bit-flip, Gaussian Strengths: global search, diverse exploration Weakness: may converge slowly

3. Simulated Annealing (SA)

Mimics cooling of metals, start hot (high randomness), slowly cool.

At each step:

1. Propose random neighbor
2. Accept if better
3. If worse, accept with probability
$$P = e^{-\frac{\Delta E}{T}}$$
4. Gradually lower (T)

Tiny Code

```
T = T_init;
state = random_state();
while (T > T_min) {
    next = neighbor(state);
    dE = cost(next) - cost(state);
    if (dE < 0 || exp(-dE/T) > rand_uniform())
        state = next;
    T *= alpha; // cooling rate
}
```

Strengths: escapes local minima Weakness: sensitive to cooling schedule

4. Particle Swarm Optimization (PSO)

Inspired by bird flocking. Each particle adjusts velocity based on:

- Its own best position- The global best found

$$v_i \leftarrow wv_i + c_1 r_1 (p_i - x_i) + c_2 r_2 (g - x_i)$$

$$x_i \leftarrow x_i + v_i$$

Tiny Code

```
for each particle i:
    v[i] = w*v[i] + c1*r1*(pbest[i]-x[i]) + c2*r2*(gbest-x[i]);
    x[i] += v[i];
    update_best(i);
```

Strengths: continuous domains, easy Weakness: premature convergence

5. Ant Colony Optimization (ACO)

Inspired by ant foraging, ants deposit pheromones on paths. The stronger the trail, the more likely others follow.

Steps:

1. Initialize pheromone on all edges
2. Each ant builds a solution (prob.   pheromone)
3. Evaluate paths
4. Evaporate pheromone
5. Reinforce good paths

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \sum_k \Delta\tau_{ij}^k$$

Tiny Code

```
for each iteration:
    for each ant:
        path = build_solution(pheromone);
        score = evaluate(path);
    evaporate(pheromone);
    deposit(pheromone, best_paths);
```

Strengths: combinatorial problems (TSP) Weakness: parameter tuning, slower convergence

6. Comparing the Four

Method	Inspiration	Best For	Key Idea
GA	Evolution	Discrete search	Selection, crossover, mutation
SA	Thermodynamics	Local optima escape	Cooling + randomness
PSO	Swarm behavior	Continuous search	Local + global attraction
ACO	Ant foraging	Graph paths	Pheromone reinforcement

7. Design Patterns

Common metaheuristic pattern:

- Represent solution- Define fitness / cost function- Define neighbor / mutation operators- Balance randomness and greediness Tuning parameters often matters more than equations.

8. Hybrid Metaheuristics

Combine strengths:

- GA + SA: evolve population, fine-tune locally- PSO + DE: use swarm + differential evolution- ACO + Local Search: reinforce with hill-climbing These hybrids often outperform single methods.

9. Common Pitfalls

- Poor representation → weak search- Over-exploitation → stuck in local optima- Bad parameters → chaotic or stagnant behavior Always visualize progress (fitness over time).

10. Why It Matters

Metaheuristics give us adaptive intelligence, searching without gradients, equations, or complete knowledge. They reflect nature's way of solving complex puzzles: iterate, adapt, survive.

> "Optimization is not about perfection. It's about progress guided by curiosity."

Try It Yourself

1. Implement Simulated Annealing for the Traveling Salesman Problem.
2. Create a Genetic Algorithm for knapsack optimization.
3. Tune PSO parameters to fit a function $f(x) = x^2 + 10 \sin x$.
4. Compare ACO paths for TSP at different evaporation rates.

97. Reinforcement Learning (Q-learning, Policy Gradients)

Reinforcement Learning (RL) is about learning through interaction , an agent explores an environment, takes actions, and learns from rewards. Unlike supervised learning (where correct labels are given), RL learns what to do by trial and error.

This section introduces two core approaches:

- Q-learning (value-based)- Policy Gradient (policy-based)

1. The Reinforcement Learning Setting

An RL problem is modeled as a Markov Decision Process (MDP):

- States S
- Actions A
- Transition $P(s' \mid s, a)$
- Reward $R(s, a)$
- Discount factor γ

The agent's goal is to find a policy $\pi(a \mid s)$ that maximizes expected return:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

2. Value Functions

The value function measures how good a state (or state-action pair) is.

- State-value:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t | S_t = s]$$

- Action-value (Q-function):

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a]$$

3. Bellman Equation

The Bellman equation relates a state's value to its neighbors:

$$Q^*(s,a) = R(s,a) + \gamma \max_{a'} Q^*(s',a')$$

This recursive definition drives value iteration and Q-learning.

4. Q-Learning

Q-learning learns the optimal action-value function off-policy (independent of behavior policy):

Update Rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Tiny Code

```
Q[s][a] += alpha * (r + gamma * max(Q[s_next]) - Q[s][a]);
s = s_next;
```

Repeat while exploring (e.g., ε-greedy):

- With probability ε, choose a random action
- With probability $1 - \varepsilon$, choose the best action

Over time, Q converges to Q^*.

5. Exploration vs Exploitation

RL is a balancing act:

- Exploration: try new actions to gather knowledge- Exploitation: use current best knowledge to maximize reward Strategies:
- -greedy- Softmax action selection- Upper Confidence Bound (UCB)

6. Policy Gradient Methods

Instead of learning Q-values, learn the policy directly. Represent policy with parameters θ:

$$\pi_\theta(a|s) = P(a|s;\theta)$$

Goal: maximize expected return

$$J(\theta) = \mathbb{E} * \pi * \theta[G_t]$$

Gradient ascent update:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

REINFORCE Algorithm:

$$\nabla_\theta J(\theta) = \mathbb{E}\big[G_t \nabla_\theta \log \pi_\theta(a_t|s_t)\big]$$

Tiny Code

```
theta += alpha * G_t * grad_logpi(a_t, s_t);
```

7. Actor-Critic Architecture

Combines policy gradient (actor) + value estimation (critic).

- Actor: updates policy- Critic: estimates value (baseline) Update:

$$\theta \leftarrow \theta + \alpha_\theta \delta_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

$$w \leftarrow w + \alpha_w \delta_t \nabla_w V_w(s_t)$$

with TD error:

$$\delta_t = r + \gamma V(s') - V(s)$$

8. Comparing Methods

Method	Type	Learns	On/Off Policy	Continuous?	
Q-learning	Value-based	Q(s, a)	Off-policy	No	
Policy Gradient	Policy-based	(a	s)	On-policy	Yes
Actor-Critic	Hybrid	Both	On-policy	Yes	

9. Extensions

- Deep Q-Networks (DQN): use neural nets for Q(s, a)- PPO / A3C: advanced actor-critic methods- TD( ): tradeoff between MC and TD learning- Double Q-learning: reduce overestimation- Entropy regularization: encourage exploration

10. Why It Matters

Reinforcement learning powers autonomous agents, game AIs, and control systems. It's the foundation of AlphaGo, robotics control, and adaptive decision systems.

> "An agent learns not from instruction but from experience."

Try It Yourself

1. Implement Q-learning for a grid-world maze.
2. Add -greedy exploration.
3. Visualize the learned policy.
4. Try REINFORCE with a simple policy (e.g. softmax over actions).
5. Compare convergence of Q-learning vs Policy Gradient.

98. Approximation and Online Algorithms

In the real world, we often can't wait for a perfect solution , data arrives on the fly, or the problem is too hard to solve exactly. That's where approximation and online algorithms shine. They aim for good-enough results, fast and adaptively, under uncertainty.

1. The Big Picture

- Approximation algorithms: Solve NP-hard problems with provable bounds.- Online algorithms: Make immediate decisions without knowing the future. Both trade optimality for efficiency or adaptability.

2. Approximation Algorithms

An approximation algorithm finds a solution within a factor ρ of the optimal.

If (C) is cost of the algorithm, and C^* is optimal cost:

$$\rho = \max\left(\frac{C}{C^*}, \frac{C^*}{C}\right)$$

Example: $\rho = 2 \rightarrow$ solution at most twice worse than optimal.

3. Example: Vertex Cover

Problem: Given graph (G(V,E)), choose smallest set of vertices covering all edges.

Algorithm (2-approximation):

1. Initialize cover =
2. While edges remain:
    - Pick any edge (u, v) - Add both u, v to cover - Remove all edges incident on u or v Guarantee: At most $2\times$ optimal size.

Tiny Code

```
cover = {};
while (!edges.empty()) {
    (u, v) = edges.pop();
    cover.add(u);
    cover.add(v);
    remove_incident_edges(u, v);
}
```

4. Example: Metric TSP (Triangle Inequality)

Algorithm (Christofides):

1. Find MST
2. Find odd-degree vertices
3. Find min perfect matching
4. Combine + shortcut to get tour

Guarantee: $1.5 \times$ optimal.

5. Greedy Approximation: Set Cover

Goal: Cover universe (U) with minimum sets S_i.

Greedy Algorithm: Pick set covering most uncovered elements each time. Guarantee: $H_n \approx \ln n$ factor approximation.

6. Online Algorithms

Online algorithms must decide now, before future input is known.

Goal: Minimize competitive ratio:

$$\text{CR} = \max_{\text{input}} \frac{\text{Cost} * \text{online}}{\text{Cost} * \text{optimal offline}}$$

Lower CR $\rightarrow$ better adaptability.

7. Classic Example: Online Paging

You have k pages in cache, sequence of page requests.

- If page in cache $\rightarrow$ hit- Else $\rightarrow$ miss, must evict one page Strategies:
- LRU (Least Recently Used): evict oldest- FIFO: evict first loaded- Random: pick randomly Competitive Ratio:
- LRU: (k)- Random: (2k-1)

Tiny Code

```
cache = LRUCache(k);
for (page in requests) {
    if (!cache.contains(page))
        cache.evict_oldest();
    cache.add(page);
}
```

8. Online Bipartite Matching (Karp-Vazirani-Vazirani)

Given offline set U and online set V (arrives one by one), match greedily. Competitive ratio: $1 - \frac{1}{e}$

Used in ad allocation and resource assignment.

9. Approximation + Online Together

Modern algorithms blend both:

- Streaming algorithms: One pass, small memory (Count-Min, reservoir sampling)- Online learning: Update models incrementally (SGD, perceptron)- Approximate dynamic programming: RL and heuristic search These are approximate online solvers , both quick and adaptive.

10. Why It Matters

Approximation algorithms give us provable near-optimal answers. Online algorithms give us real-time adaptivity. Together, they model intelligence under limits , when time and information are scarce.

> "Sometimes, good and on time beats perfect and late."

Try It Yourself

1. Implement 2-approx vertex cover on a small graph.
2. Simulate online paging with LRU vs Random.
3. Build a greedy set cover solver.
4. Measure competitive ratio on test sequences.
5. Combine ideas: streaming + approximation for big data filtering.

99. Fairness, Causal Inference, and Robust Optimization

As algorithms increasingly shape decisions , from hiring to lending to healthcare , we must ensure they're fair, causally sound, and robust to uncertainty. This section blends ideas from ethics, statistics, and optimization to make algorithms not just efficient, but responsible and reliable.

1. Why Fairness Matters

Machine learning systems often inherit biases from data. Without intervention, they can amplify inequality or discrimination.

Fairness-aware algorithms explicitly measure and correct these effects.

Common sources of bias:

- Historical bias (biased data)- Measurement bias (imprecise features)- Selection bias (skewed samples) The goal: equitable treatment across sensitive groups (gender, race, region, etc.)

2. Formal Fairness Criteria

Several fairness notions exist, often conflicting:

Criterion	Description	Example
Demographic Parity	($P\hat{Y} = 1 \| A = a =$ $P\hat{Y} = 1 \| A = b$)	Equal positive rate
Equal Opportunity	Equal true positive rates	Same recall for all groups
Equalized Odds	Equal TPR & FPR	Balanced errors
Calibration	Same predicted probability meaning	If model says 70%, all groups should achieve 70%

No single measure fits all , fairness depends on context and trade-offs.

3. Algorithmic Fairness Techniques

1. Pre-processing Rebalance or reweight data before training. Example: reweighing, sampling.
2. In-processing Add fairness constraints to loss function. Example: adversarial debiasing.
3. Post-processing Adjust predictions after training. Example: threshold shifting.

Tiny Code (Adversarial Debiasing Skeleton)

```
for x, a, y in data:
    y_pred = model(x)
    loss_main = loss_fn(y_pred, y)
    loss_adv = adv_fn(y_pred, a)
    loss_total = loss_main -   * loss_adv
    update(loss_total)
```

Here, the adversary tries to predict sensitive attribute, encouraging invariance.

4. Causal Inference Basics

Correlation causation. To reason about fairness and robustness, we need causal understanding , what *would* happen if we changed something.

Causal inference models relationships via Directed Acyclic Graphs (DAGs):

- Nodes: variables- Edges: causal influence

5. Counterfactual Reasoning

A counterfactual asks:

> "What would the outcome be if we intervened differently?"

Formally:

$$P(Y_{do(X=x)})$$

Used in:

- Fairness (counterfactual fairness)- Policy evaluation- Robust decision making

6. Counterfactual Fairness

An algorithm is counterfactually fair if prediction stays the same under hypothetical changes to sensitive attributes.

$$\hat{Y} * A \leftarrow a(U) = \hat{Y} * A \leftarrow a'(U)$$

This requires causal models , not just data.

7. Robust Optimization

In uncertain environments, we want solutions that hold up under worst-case conditions.

Formulation:

$$\min_{x} \max_{\xi \in \Xi} f(x, \xi)$$

where Ξ is the uncertainty set.

Example: Design a portfolio that performs well under varying market conditions.

Tiny Code

```
double robust_objective(double x[], Scenario Xi[], int N) {
    double worst = -INF;
    for (i=0; i<N; i++)
        worst = max(worst, f(x, Xi[i]));
    return worst;
}
```

This searches for a solution minimizing worst-case loss.

8. Distributional Robustness

Instead of worst-case instances, protect against worst-case distributions:

$$\min_{\theta} \sup_{Q \in \mathcal{B}(P)} \mathbb{E}_{x \sim Q}[L(\theta, x)]$$

Used in adversarial training and domain adaptation.

Example: Add noise or perturbations to improve resilience:

```
x_adv = x +   * sign(grad(loss, x))
```

9. Balancing Fairness, Causality, and Robustness

Goal	Method	Challenge
Fairness	Parity, Adversarial, Counterfactual	Competing definitions
Causality	DAGs, do-calculus, SCMs	Identifying true structure
Robustness	Min-max, DRO, Adversarial Training	Trade-off with accuracy

Real-world design involves balancing trade-offs.

Sometimes improving fairness reduces accuracy, or robustness increases conservatism.

10. Why It Matters

Algorithms don't exist in isolation , they affect people. Embedding fairness, causality, and robustness ensures systems are trustworthy, interpretable, and just.

> "The goal is not just intelligent algorithms , but responsible ones."

Try It Yourself

1. Train a simple classifier on biased data.
2. Apply reweighing or adversarial debiasing.
3. Draw a causal DAG of your data features.
4. Compute counterfactual fairness for a sample.
5. Implement a robust loss using adversarial perturbations.

100. AI Planning, Search, and Learning Systems

AI systems are not just pattern recognizers , they are decision makers. They plan, search, and learn in structured environments, choosing actions that lead to long-term goals. This section explores how modern AI combines planning, search, and learning to solve complex tasks.

1. What Is AI Planning?

AI planning is about finding a sequence of actions that transforms an initial state into a goal state.

Formally, a planning problem consists of:

- States (S)- Actions (A)- Transition function (T(s, a) →s')- Goal condition $G \subseteq S$- Cost function (c(a)) The objective: Find a plan $\pi = [a_1, a_2, \dots, a_n]$ minimizing total cost or maximizing reward.

2. Search-Based Planning

At the heart of planning lies search. Search explores possible action sequences, guided by heuristics.

Algorithm	Type	Description
DFS	Uninformed	Deep exploration, no guarantee
BFS	Uninformed	Finds shortest path
Dijkstra	Weighted	Optimal if costs 0
A*	Heuristic	Combines cost + heuristic

A* Search Formula:

$$f(n) = g(n) + h(n)$$

where:

- (g(n)): cost so far- (h(n)): heuristic estimate to goal If (h) is admissible, A* is optimal.

Tiny Code (A* Skeleton)

```
priority_queue<Node> open;
g[start] = 0;
open.push({start, h(start)});

while (!open.empty()) {
    n = open.pop_min();
    if (goal(n)) break;
    for (a in actions(n)) {
        s = step(n, a);
        cost = g[n] + c(n, a);
        if (cost < g[s]) {
            g[s] = cost;
            f[s] = g[s] + h(s);
            open.push({s, f[s]});
        }
    }
}
```

3. Heuristics and Admissibility

A heuristic (h(s)) estimates distance to the goal.

- Admissible: never overestimates- Consistent: satisfies triangle inequality Examples:
- Manhattan distance (grids)- Euclidean distance (geometry)- Pattern databases (puzzles) Good heuristics = faster convergence.

4. Classical Planning (STRIPS)

In symbolic AI, states are represented by facts (predicates), and actions have preconditions and effects.

Example:

```
Action: Move(x, y)
Precondition: At(x), Clear(y)
Effect: ¬At(x), At(y)
```

Search happens in logical state space.

Planners:

- Forward search (progression)- Backward search (regression)- Heuristic planners (FF, HSP)

5. Hierarchical Planning

Break complex goals into subgoals.

- HTN (Hierarchical Task Networks): Define high-level tasks broken into subtasks.

Example: “Make dinner” → [Cook rice, Stir-fry vegetables, Set table]

Hierarchy makes planning modular and interpretable.

6. Probabilistic Planning

When actions are uncertain:

- MDPs: full observability, stochastic transitions- POMDPs: partial observability Use value iteration, policy iteration, or Monte Carlo planning.

7. Learning to Plan

Combine learning with search:

- Learned heuristics: neural networks approximate (h(s))- AlphaZero-style planning: learn value + policy, guide tree search- Imitation learning: mimic expert demonstrations This bridges classical AI and modern ML.

Tiny Code (Learning-Guided A*)

```
f = g + alpha * learned_heuristic(s)
```

Neural net learns (h_ (s)) from solved examples.

8. Integrated Systems

Modern AI stacks combine:

- Search (planning backbone)- Learning (policy, heuristic, model)- Simulation (data generation) Examples:
- AlphaZero: self-play + MCTS + neural nets- MuZero: learns model + value + policy jointly- Large Language Agents: use reasoning + memory + search

9. Real-World Applications

- Robotics: motion planning, pathfinding- Games: Go, Chess, strategy games- Logistics: route optimization- Autonomy: drones, vehicles, AI assistants- Synthesis: program and query generation Each blends symbolic reasoning and statistical learning.

10. Why It Matters

Planning, search, and learning form the triad of intelligence:

- Search explores possibilities- Planning sequences actions toward goals- Learning adapts heuristics from experience Together, they power systems that think, adapt, and act.

  "Intelligence is not just knowing , it is choosing wisely under constraints."

Try It Yourself

1. Implement A* search on a grid maze.
2. Add a Manhattan heuristic.
3. Extend to probabilistic transitions (simulate noise).
4. Build a simple planner with preconditions and effects.
5. Train a neural heuristic to guide search on puzzles.

The Plan

Chapter 1. Foundations of Algorithms

1. What Is an Algorithm?

#	Algorithm	Note
1	Euclid's GCD	Oldest known algorithm for greatest common divisor
2	Sieve of Eratosthenes	Generate primes efficiently
3	Binary Search	Divide and conquer search
4	Exponentiation by Squaring	Fast power computation
5	Long Division	Classic step-by-step arithmetic
6	Modular Addition Algorithm	Wrap-around arithmetic
7	Base Conversion Algorithm	Convert between number systems
8	Factorial Computation	Recursive and iterative approaches
9	Fibonacci Sequence	Recursive vs. dynamic computation
10	Tower of Hanoi	Recursive problem-solving pattern

2. Measuring Time and Space

#	Algorithm	Note
11	Counting Operations	Manual step-counting for complexity
12	Loop Analysis	Evaluate time cost of loops
13	Recurrence Expansion	Analyze recursive costs
14	Amortized Analysis	Average per-operation cost
15	Space Counting	Stack and heap tracking
16	Memory Footprint Estimator	Track per-variable usage
17	Time Complexity Table	Map O(1)…O(n^2)…O(2)
18	Space-Time Tradeoff	Cache vs. recomputation
19	Profiling Algorithm	Empirical time measurement
20	Benchmarking Framework	Compare algorithm performance

3. Big-O, Big-Theta, Big-Omega

#	Algorithm	Note
21	Growth Rate Comparator	Compare asymptotic behaviors
22	Dominant Term Extractor	Simplify runtime expressions
23	Limit-Based Complexity Test	Using limits for asymptotics
24	Summation Simplifier	Sum of arithmetic/geometric sequences
25	Recurrence Tree Method	Visualize recursive costs
26	Master Theorem Evaluator	Solve T(n) recurrences
27	Big-Theta Proof Builder	Bounding upper and lower limits
28	Big-Omega Case Finder	Best-case scenario analysis
29	Empirical Complexity Estimator	Measure via doubling experiments
30	Complexity Class Identifier	Match runtime to known class

4. Algorithmic Paradigms (Greedy, Divide and Conquer, DP)

#	Algorithm	Note
31	Greedy Coin Change	Local optimal step-by-step
32	Huffman Coding	Greedy compression tree
33	Merge Sort	Divide and conquer sort
34	Binary Search	Divide and conquer search
35	Karatsuba Multiplication	Recursive divide & conquer
36	Matrix Chain Multiplication	DP with substructure
37	Longest Common Subsequence	Classic DP problem
38	Rod Cutting	DP optimization
39	Activity Selection	Greedy scheduling
40	Optimal Merge Patterns	Greedy file merging

5. Recurrence Relations

#	Algorithm	Note
41	Linear Recurrence Solver	Closed-form for linear recurrences
42	Master Theorem	Divide-and-conquer complexity
43	Substitution Method	Inductive proof approach
44	Iteration Method	Expand recurrence step-by-step
45	Generating Functions	Transform recurrences
46	Matrix Exponentiation	Solve linear recurrences fast

#	Algorithm	Note
47	Recurrence to DP Table	Tabulation approach
48	Divide & Combine Template	Convert recurrence into algorithm
49	Memoized Recursive Solver	Store overlapping results
50	Characteristic Polynomial	Solve homogeneous recurrence

6. Searching Basics

#	Algorithm	Note
51	Linear Search	Sequential element scan
52	Binary Search	Midpoint halving
53	Jump Search	Block skip linear
54	Exponential Search	Doubling step size
55	Interpolation Search	Estimate position by value
56	Ternary Search	Divide into thirds
57	Fibonacci Search	Golden ratio search
58	Sentinel Search	Early termination optimization
59	Bidirectional Search	Meet-in-the-middle
60	Search in Rotated Array	Adapted binary search

7. Sorting Basics

#	Algorithm	Note
61	Bubble Sort	Adjacent swap sort
62	Selection Sort	Find minimum each pass
63	Insertion Sort	Incremental build sort
64	Shell Sort	Gap-based insertion
65	Merge Sort	Divide-and-conquer
66	Quick Sort	Partition-based
67	Heap Sort	Binary heap order
68	Counting Sort	Integer key distribution
69	Radix Sort	Digit-by-digit
70	Bucket Sort	Group into ranges

8. Data Structures Overview

#	Algorithm	Note
71	Stack Push/Pop	LIFO operations
72	Queue Enqueue/Dequeue	FIFO operations
73	Singly Linked List	Linear node chain
74	Doubly Linked List	Bidirectional traversal
75	Hash Table Insertion	Key-value indexing
76	Binary Search Tree Insert	Ordered node placement
77	Heapify	Build heap in-place
78	Union-Find Operations	Disjoint-set management
79	Graph Adjacency List Build	Sparse representation
80	Trie Insertion/Search	Prefix tree for strings

9. Graphs and Trees Overview

#	Algorithm	Note
81	DFS Traversal	Depth-first exploration
82	BFS Traversal	Level-order exploration
83	Topological Sort	DAG ordering
84	Minimum Spanning Tree	Kruskal/Prim overview
85	Dijkstra's Shortest Path	Weighted graph shortest route
86	Bellman-Ford	Handle negative edges
87	Floyd-Warshall	All-pairs shortest path
88	Union-Find for MST	Edge grouping
89	Tree Traversals	Inorder, Preorder, Postorder
90	LCA (Lowest Common Ancestor)	Common node in tree

10. Algorithm Design Patterns

#	Algorithm	Note
91	Brute Force	Try all possibilities
92	Greedy Choice	Local optimum per step
93	Divide and Conquer	Break and merge
94	Dynamic Programming	Reuse subproblems
95	Backtracking	Explore with undo
96	Branch and Bound	Prune search space
97	Randomized Algorithm	Inject randomness
98	Approximation Algorithm	Near-optimal solution
99	Online Algorithm	Step-by-step decision

#	Algorithm	Note
100	Hybrid Strategy	Combine paradigms

Chapter 2. Sorting and Searching

11. Elementary Sorting (Bubble, Insertion, Selection)

#	Algorithm	Note
101	Bubble Sort	Swap adjacent out-of-order elements
102	Improved Bubble Sort	Stop early if already sorted
103	Cocktail Shaker Sort	Bidirectional bubble pass
104	Selection Sort	Select smallest element each pass
105	Double Selection Sort	Find both min and max each pass
106	Insertion Sort	Insert each element into correct spot
107	Binary Insertion Sort	Use binary search for position
108	Gnome Sort	Simple insertion-like with swaps
109	Odd-Even Sort	Parallel-friendly comparison sort
110	Stooge Sort	Recursive quirky educational sort

12. Divide-and-Conquer Sorting (Merge, Quick, Heap)

#	Algorithm	Note
111	Merge Sort	Recursive divide and merge
112	Iterative Merge Sort	Bottom-up non-recursive version
113	Quick Sort	Partition-based recursive sort
114	Hoare Partition Scheme	Classic quicksort partition
115	Lomuto Partition Scheme	Simpler but less efficient
116	Randomized Quick Sort	Avoid worst-case pivot
117	Heap Sort	Heapify + extract max repeatedly
118	3-Way Quick Sort	Handle duplicates efficiently
119	External Merge Sort	Disk-based merge for large data
120	Parallel Merge Sort	Divide work among threads

13. Counting and Distribution Sorts (Counting, Radix, Bucket)

#	Algorithm	Note
121	Counting Sort	Count key occurrences
122	Stable Counting Sort	Preserve order of equals
123	Radix Sort (LSD)	Least significant digit first
124	Radix Sort (MSD)	Most significant digit first
125	Bucket Sort	Distribute into buckets
126	Pigeonhole Sort	Simple bucket variant
127	Flash Sort	Distribution with in-place correction
128	Postman Sort	Stable multi-key sort
129	Address Calculation Sort	Hash-like distribution
130	Spread Sort	Hybrid radix/quick strategy

14. Hybrid Sorts (IntroSort, Timsort)

#	Algorithm	Note
131	IntroSort	Quick + Heap fallback
132	TimSort	Merge + Insertion + Runs
133	Dual-Pivot QuickSort	Modern quicksort optimization
134	SmoothSort	Heap-like adaptive sort
135	Block Merge Sort	Cache-efficient merge variant
136	Adaptive Merge Sort	Adjusts to partially sorted data
137	PDQSort	Pattern-defeating quicksort
138	WikiSort	Stable in-place merge
139	GrailSort	In-place stable mergesort
140	Adaptive Hybrid Sort	Dynamically selects strategy

15. Special Sorts (Cycle, Gnome, Comb, Pancake)

#	Algorithm	Note
141	Cycle Sort	Minimal writes
142	Comb Sort	Shrinking gap bubble
143	Gnome Sort	Insertion-like with swaps
144	Cocktail Sort	Two-way bubble
145	Pancake Sort	Flip-based sorting
146	Bitonic Sort	Parallel network sorting
147	Odd-Even Merge Sort	Sorting network design
148	Sleep Sort	Uses timing as order key
149	Bead Sort	Simulates gravity

#	Algorithm	Note
150	Bogo Sort	Randomly permute until sorted

16. Linear and Binary Search

#	Algorithm	Note
151	Linear Search	Scan sequentially
152	Linear Search (Sentinel)	Guard element at end
153	Binary Search (Iterative)	Halve interval each loop
154	Binary Search (Recursive)	Halve interval via recursion
155	Binary Search (Lower Bound)	First >= target
156	Binary Search (Upper Bound)	First > target
157	Exponential Search	Double step size
158	Jump Search	Jump fixed steps then linear
159	Fibonacci Search	Golden-ratio style jumps
160	Uniform Binary Search	Avoid recomputing midpoints

17. Interpolation and Exponential Search

#	Algorithm	Note
161	Interpolation Search	Estimate index by value
162	Recursive Interpolation Search	Divide by estimated midpoint
163	Exponential Search	Double and binary refine
164	Doubling Search	Generic exponential pattern
165	Galloping Search	Used in TimSort merges
166	Unbounded Binary Search	Find bounds dynamically
167	Root-Finding Bisection	Search zero-crossing
168	Golden Section Search	Optimize unimodal function
169	Fibonacci Search (Optimum)	Similar to golden search
170	Jump + Binary Hybrid	Combined probing strategy

18. Selection Algorithms (Quickselect, Median of Medians)

#	Algorithm	Note
171	Quickselect	Partition-based selection

#	Algorithm	Note
172	Median of Medians	Deterministic pivot
173	Randomized Select	Random pivot version
174	Binary Search on Answer	Range-based selection
175	Order Statistics Tree	BST with rank queries
176	Tournament Tree Selection	Hierarchical comparison
177	Heap Select (Min-Heap)	Maintain top-k elements
178	Partial QuickSort	Sort partial prefix
179	BFPRT Algorithm	Linear-time selection
180	Kth Largest Stream	Streaming selection

19. Range Searching and Nearest Neighbor

#	Algorithm	Note
181	Binary Search Range	Find lower and upper bounds
182	Segment Tree Query	Sum/min/max over interval
183	Fenwick Tree Query	Efficient prefix sums
184	Interval Tree Search	Overlap queries
185	KD-Tree Search	Spatial nearest neighbor
186	R-Tree Query	Range search in geometry
187	Range Minimum Query (RMQ)	Sparse table approach
188	Mo's Algorithm	Offline query reordering
189	Sweep Line Range Search	Sort + scan technique
190	Ball Tree Nearest Neighbor	Metric-space search

20. Search Optimizations and Variants

#	Algorithm	Note
191	Binary Search with Tolerance	For floating values
192	Ternary Search	Unimodal optimization
193	Hash-Based Search	O(1) expected lookup
194	Bloom Filter Lookup	Probabilistic membership
195	Cuckoo Hash Search	Dual-hash relocation
196	Robin Hood Hashing	Equalize probe lengths
197	Jump Consistent Hashing	Stable hash assignment
198	Prefix Search in Trie	Auto-completion lookup
199	Pattern Search in Suffix Array	Fast substring lookup
200	Search in Infinite Array	Dynamic bound finding

#	Algorithm	Note

Chapter 3. Data Structures in Action

21. Arrays, Linked Lists, Stacks, Queues

#	Algorithm	Note
201	Dynamic Array Resize	Doubling strategy for capacity
202	Circular Array Implementation	Wrap-around indexing
203	Singly Linked List Insert/Delete	Basic node manipulation
204	Doubly Linked List Insert/Delete	Two-way linkage
205	Stack Push/Pop	LIFO structure
206	Queue Enqueue/Dequeue	FIFO structure
207	Deque Implementation	Double-ended queue
208	Circular Queue	Fixed-size queue with wrap-around
209	Stack via Queue	Implement stack using two queues
210	Queue via Stack	Implement queue using two stacks

22. Hash Tables and Variants (Cuckoo, Robin Hood, Consistent)

#	Algorithm	Note
211	Hash Table Insertion	Key-value pair with modulo
212	Linear Probing	Resolve collisions sequentially
213	Quadratic Probing	Nonlinear probing sequence
214	Double Hashing	Alternate hash on collision
215	Cuckoo Hashing	Two-table relocation strategy
216	Robin Hood Hashing	Equalize probe length fairness
217	Chained Hash Table	Linked list buckets
218	Perfect Hashing	No-collision mapping
219	Consistent Hashing	Stable distribution across nodes
220	Dynamic Rehashing	Resize on load factor threshold

23. Heaps (Binary, Fibonacci, Pairing)

#	Algorithm	Note
221	Binary Heap Insert	Bubble-up maintenance
222	Binary Heap Delete	Heapify-down maintenance
223	Build Heap (Heapify)	Bottom-up O(n) build
224	Heap Sort	Extract max repeatedly
225	Min Heap Implementation	For smallest element access
226	Max Heap Implementation	For largest element access
227	Fibonacci Heap Insert/Delete	Amortized efficient operations
228	Pairing Heap Merge	Lightweight mergeable heap
229	Binomial Heap Merge	Merge trees of equal order
230	Leftist Heap Merge	Maintain rank-skewed heap

24. Balanced Trees (AVL, Red-Black, Splay, Treap)

#	Algorithm	Note
231	AVL Tree Insert	Rotate to maintain balance
232	AVL Tree Delete	Balance after deletion
233	Red-Black Tree Insert	Color fix and rotations
234	Red-Black Tree Delete	Maintain invariants
235	Splay Tree Access	Move accessed node to root
236	Treap Insert	Priority-based rotation
237	Treap Delete	Randomized balance
238	Weight Balanced Tree	Maintain subtree weights
239	Scapegoat Tree Rebuild	Rebalance on size threshold
240	AA Tree	Simplified red-black variant

25. Segment Trees and Fenwick Trees

#	Algorithm	Note
241	Build Segment Tree	Recursive construction
242	Range Sum Query	Recursive or iterative query
243	Range Update	Lazy propagation technique
244	Point Update	Modify single element
245	Fenwick Tree Build	Incremental binary index
246	Fenwick Update	Update cumulative sums
247	Fenwick Query	Prefix sum retrieval
248	Segment Tree Merge	Combine child results
249	Persistent Segment Tree	Maintain history of versions

#	Algorithm	Note
250	2D Segment Tree	For matrix range queries

26. Disjoint Set Union (Union-Find)

#	Algorithm	Note
251	Make-Set	Initialize each element
252	Find	Locate representative
253	Union	Merge two sets
254	Union by Rank	Attach smaller tree to larger
255	Path Compression	Flatten tree structure
256	DSU with Rollback	Support undo operations
257	DSU on Tree	Track subtree connectivity
258	Kruskal's MST	Edge selection with DSU
259	Connected Components	Group graph nodes
260	Offline Query DSU	Handle dynamic unions

27. Probabilistic Data Structures (Bloom, Count-Min, HyperLogLog)

#	Algorithm	Note
261	Bloom Filter Insert	Hash to bit array
262	Bloom Filter Query	Probabilistic membership check
263	Counting Bloom Filter	Support deletions via counters
264	Cuckoo Filter	Space-efficient alternative
265	Count-Min Sketch	Approximate frequency table
266	HyperLogLog	Cardinality estimation
267	Flajolet-Martin	Early probabilistic counting
268	MinHash	Estimate Jaccard similarity
269	Reservoir Sampling	Random k-sample stream
270	Skip Bloom Filter	Range queries on Bloom

28. Skip Lists and B-Trees

#	Algorithm	Note
271	Skip List Insert	Probabilistic layered list

#	Algorithm	Note
272	Skip List Delete	Adjust pointers
273	Skip List Search	Jump via tower levels
274	B-Tree Insert	Split on overflow
275	B-Tree Delete	Merge on underflow
276	B+ Tree Search	Leaf-based sequential scan
277	B+ Tree Range Query	Efficient ordered access
278	B* Tree	More space-efficient variant
279	Adaptive Radix Tree	Byte-wise branching
280	Trie Compression	Path compression optimization

29. Persistent and Functional Data Structures

#	Algorithm	Note
281	Persistent Stack	Keep all versions
282	Persistent Array	Copy-on-write segments
283	Persistent Segment Tree	Versioned updates
284	Persistent Linked List	Immutable nodes
285	Functional Queue	Amortized reverse lists
286	Finger Tree	Fast concat and split
287	Zipper Structure	Localized mutation
288	Red-Black Persistent Tree	Immutable balanced tree
289	Trie with Versioning	Historical string lookup
290	Persistent Union-Find	Time-travel connectivity

30. Advanced Trees and Range Queries

#	Algorithm	Note
291	Sparse Table Build	Static range min/max
292	Cartesian Tree	RMQ to LCA transformation
293	Segment Tree Beats	Handle complex queries
294	Merge Sort Tree	Range count queries
295	Wavelet Tree	Rank/select by value
296	KD-Tree	Multidimensional queries
297	Range Tree	Orthogonal range queries
298	Fenwick 2D Tree	Matrix prefix sums
299	Treap Split/Merge	Range-based treap ops
300	Mo's Algorithm on Tree	Offline subtree queries

Chapter 4. Graph Algorithms

31. Traversals (DFS, BFS, Iterative Deepening)

#	Algorithm	Note
301	Depth-First Search (Recursive)	Explore deeply before backtracking
302	Depth-First Search (Iterative)	Stack-based exploration
303	Breadth-First Search (Queue)	Level-order traversal
304	Iterative Deepening DFS	Combine depth-limit + completeness
305	Bidirectional BFS	Search from both ends
306	DFS on Grid	Maze solving / connected components
307	BFS on Grid	Shortest path in unweighted graph
308	Multi-Source BFS	Parallel layer expansion
309	Topological Sort (DFS-based)	DAG ordering
310	Topological Sort (Kahn's Algorithm)	In-degree tracking

32. Strongly Connected Components (Tarjan, Kosaraju)

#	Algorithm	Note
311	Kosaraju's Algorithm	Two-pass DFS
312	Tarjan's Algorithm	Low-link discovery
313	Gabow's Algorithm	Stack pair tracking
314	SCC DAG Construction	Condensed component graph
315	SCC Online Merge	Incremental condensation
316	Component Label Propagation	Iterative labeling
317	Path-Based SCC	DFS with path stack
318	Kosaraju Parallel Version	SCC via parallel DFS
319	Dynamic SCC Maintenance	Add/remove edges
320	SCC for Weighted Graph	Combine with edge weights

33. Shortest Paths (Dijkstra, Bellman-Ford, A*, Johnson)

#	Algorithm	Note
321	Dijkstra (Binary Heap)	Greedy edge relaxation
322	Dijkstra (Fibonacci Heap)	Improved priority queue
323	Bellman-Ford	Negative weights support
324	SPFA (Queue Optimization)	Faster average Bellman-Ford

#	Algorithm	Note
325	A* Search	Heuristic-guided path
326	Floyd–Warshall	All-pairs shortest path
327	Johnson's Algorithm	All-pairs using reweighting
328	0-1 BFS	Deque-based shortest path
329	Dial's Algorithm	Integer weight buckets
330	Multi-Source Dijkstra	Multiple starting points

34. Shortest Path Variants (0–1 BFS, Bidirectional, Heuristic A*)

#	Algorithm	Note
331	0–1 BFS	For edges with weight 0 or 1
332	Bidirectional Dijkstra	Meet in the middle
333	A* with Euclidean Heuristic	Spatial shortest path
334	ALT Algorithm	A* landmarks + triangle inequality
335	Contraction Hierarchies	Preprocessing for road networks
336	CH Query Algorithm	Shortcut-based routing
337	Bellman-Ford Queue Variant	Early termination
338	Dijkstra with Early Stop	Halt on target
339	Goal-Directed Search	Restrict expansion direction
340	Yen's K Shortest Paths	Enumerate multiple best paths

35. Minimum Spanning Trees (Kruskal, Prim, Borůvka)

#	Algorithm	Note
341	Kruskal's Algorithm	Sort edges + union-find
342	Prim's Algorithm (Heap)	Grow MST from seed
343	Prim's Algorithm (Adj Matrix)	Dense graph variant
344	Borůvka's Algorithm	Component merging
345	Reverse-Delete MST	Remove heavy edges
346	MST via Dijkstra Trick	For positive weights
347	Dynamic MST Maintenance	Handle edge updates
348	Minimum Bottleneck Spanning Tree	Max edge minimization
349	Manhattan MST	Grid graph optimization
350	Euclidean MST (Kruskal + Geometry)	Use Delaunay graph

36. Flows (Ford–Fulkerson, Edmonds–Karp, Dinic)

#	Algorithm	Note
351	Ford–Fulkerson	Augmenting path method
352	Edmonds–Karp	BFS-based Ford–Fulkerson
353	Dinic's Algorithm	Level graph + blocking flow
354	Push–Relabel	Local preflow push
355	Capacity Scaling	Speed-up with capacity tiers
356	Cost Scaling	Min-cost optimization
357	Min-Cost Max-Flow (Bellman-Ford)	Costed augmenting paths
358	Min-Cost Max-Flow (SPFA)	Faster average
359	Circulation with Demands	Generalized flow formulation
360	Successive Shortest Path	Incremental min-cost updates

37. Cuts (Stoer–Wagner, Karger, Gomory–Hu)

#	Algorithm	Note
361	Stoer–Wagner Minimum Cut	Global min cut
362	Karger's Randomized Cut	Contract edges randomly
363	Karger–Stein	Recursive randomized cut
364	Gomory–Hu Tree	All-pairs min-cut
365	Max-Flow Min-Cut	Duality theorem application
366	Stoer–Wagner Repeated Phase	Multiple passes
367	Dynamic Min Cut	Maintain on edge update
368	Minimum s–t Cut (Edmonds–Karp)	Based on flow
369	Approximate Min Cut	Random sampling
370	Min k-Cut	Partition graph into k parts

38. Matchings (Hopcroft–Karp, Hungarian, Blossom)

#	Algorithm	Note
371	Bipartite Matching (DFS)	Simple augmenting path
372	Hopcroft–Karp	$O(E\sqrt{V})$ bipartite matching
373	Hungarian Algorithm	Weighted assignment
374	Kuhn–Munkres	Max-weight matching
375	Blossom Algorithm	General graph matching
376	Edmonds' Blossom Shrinking	Odd cycle contraction

#	Algorithm	Note
377	Greedy Matching	Fast approximate
378	Stable Marriage (Gale–Shapley)	Stable pairing
379	Weighted b-Matching	Capacity-constrained
380	Maximal Matching	Local greedy maximal set

39. Tree Algorithms (LCA, HLD, Centroid Decomposition)

#	Algorithm	Note
381	Euler Tour LCA	Flatten tree to array
382	Binary Lifting LCA	Jump powers of two
383	Tarjan's LCA (Offline DSU)	Query via union-find
384	Heavy-Light Decomposition	Decompose paths
385	Centroid Decomposition	Recursive split on centroid
386	Tree Diameter (DFS Twice)	Farthest pair
387	Tree DP	Subtree-based optimization
388	Rerooting DP	Compute all roots' answers
389	Binary Search on Tree	Edge weight constraints
390	Virtual Tree	Build on query subset

40. Advanced Graph Algorithms and Tricks

#	Algorithm	Note
391	Topological DP	DP on DAG order
392	SCC Condensed Graph DP	Meta-graph processing
393	Eulerian Path	Trail covering all edges
394	Hamiltonian Path	NP-complete exploration
395	Chinese Postman	Eulerian circuit with repeats
396	Hierholzer's Algorithm	Construct Eulerian circuit
397	Johnson's Cycle Finding	Enumerate all cycles
398	Transitive Closure (Floyd–Warshall)	Reachability matrix
399	Graph Coloring (Backtracking)	Constraint satisfaction
400	Articulation Points & Bridges	Critical structure detection

Chapter 5. Dynamic Programming

41. DP Basics and State Transitions

#	Algorithm	Note
401	Fibonacci DP	Classic top-down vs bottom-up
402	Climbing Stairs	Count paths with small steps
403	Grid Paths	DP over 2D lattice
404	Min Cost Path	Accumulate minimal sums
405	Coin Change (Count Ways)	Combinatorial sums
406	Coin Change (Min Coins)	Minimize step count
407	Knapsack 0/1	Select items under weight limit
408	Knapsack Unbounded	Repeatable items
409	Longest Increasing Subsequence (DP)	Subsequence optimization
410	Edit Distance (Levenshtein)	Measure similarity step-by-step

42. Classic Problems (Knapsack, Subset Sum, Coin Change)

#	Algorithm	Note
411	0/1 Knapsack	Value maximization under capacity
412	Subset Sum	Boolean feasibility DP
413	Equal Partition	Divide set into equal halves
414	Count of Subsets with Sum	Counting variant
415	Target Sum	DP with +/- transitions
416	Unbounded Knapsack	Reuse items
417	Fractional Knapsack	Greedy + DP comparison
418	Coin Change (Min Coins)	DP shortest path
419	Coin Change (Count Ways)	Combinatorial counting
420	Multi-Dimensional Knapsack	Capacity in multiple dimensions

43. Sequence Problems (LIS, LCS, Edit Distance)

#	Algorithm	Note
421	Longest Increasing Subsequence	O(n^2) DP
422	LIS (Patience Sorting)	O(n log n) optimized
423	Longest Common Subsequence	Two-sequence DP
424	Edit Distance (Levenshtein)	Transform operations

#	Algorithm	Note
425	Longest Palindromic Subsequence	Symmetric DP
426	Shortest Common Supersequence	Merge sequences
427	Longest Repeated Subsequence	DP with overlap
428	String Interleaving	Merge with order preservation
429	Sequence Alignment (Bioinformatics)	Gap penalties
430	Diff Algorithm (Myers/DP)	Minimal edit path

44. Matrix and Chain Problems

#	Algorithm	Note
431	Matrix Chain Multiplication	Parenthesization cost
432	Boolean Parenthesization	Count true outcomes
433	Burst Balloons	Interval DP
434	Optimal BST	Weighted search cost
435	Polygon Triangulation	DP over partitions
436	Matrix Path Sum	DP on 2D grid
437	Largest Square Submatrix	Dynamic growth check
438	Max Rectangle in Binary Matrix	Histogram + DP
439	Submatrix Sum Queries	Prefix sum DP
440	Palindrome Partitioning	DP with cuts

45. Bitmask DP and Traveling Salesman

#	Algorithm	Note
441	Traveling Salesman (TSP)	Visit all cities
442	Subset DP	Over subsets of states
443	Hamiltonian Path DP	State compression
444	Assignment Problem DP	Mask over tasks
445	Partition into Two Sets	Balanced load
446	Count Hamiltonian Cycles	Bitmask enumeration
447	Steiner Tree DP	Minimal connection of terminals
448	SOS DP (Sum Over Subsets)	Precompute sums
449	Bitmask Knapsack	State compression
450	Bitmask Independent Set	Graph subset optimization

46. Digit DP and SOS DP

#	Algorithm	Note
451	Count Numbers with Property	Digit-state transitions
452	Count Without Adjacent Duplicates	Adjacent constraints
453	Sum of Digits in Range	Carry-dependent states
454	Count with Mod Condition	DP over digit sum mod M
455	Count of Increasing Digits	Ordered constraint
456	Count with Forbidden Digits	Exclusion transitions
457	SOS DP Subset Sum	Sum over bitmask subsets
458	SOS DP Superset Sum	Sum over supersets
459	XOR Basis DP	Combine digit and bit DP
460	Digit DP for Palindromes	Symmetric digit state

47. DP Optimizations (Divide & Conquer, Convex Hull Trick, Knuth)

#	Algorithm	Note
461	Divide & Conquer DP	Monotone decision property
462	Knuth Optimization	DP with quadrangle inequality
463	Convex Hull Trick	Linear recurrence min queries
464	Li Chao Tree	Segment-based hull maintenance
465	Slope Trick	Piecewise-linear optimization
466	Monotonic Queue Optimization	Sliding DP state
467	Bitset DP	Speed via bit-parallel
468	Offline DP Queries	Preprocessing state
469	DP + Segment Tree	Range-based optimization
470	Divide & Conquer Knapsack	Split-space DP

48. Tree DP and Rerooting

#	Algorithm	Note
471	Subtree Sum DP	Aggregate values
472	Diameter DP	Max path via child states
473	Independent Set DP	Choose or skip nodes
474	Vertex Cover DP	Tree constraint problem
475	Path Counting DP	Count root-leaf paths
476	DP on Rooted Tree	Bottom-up aggregation

#	Algorithm	Note
477	Rerooting Technique	Compute for all roots
478	Distance Sum Rerooting	Efficient recomputation
479	Tree Coloring DP	Combinatorial counting
480	Binary Search on Tree DP	Monotonic transitions

49. DP Reconstruction and Traceback

#	Algorithm	Note
481	Reconstruct LCS	Backtrack table
482	Reconstruct LIS	Track predecessors
483	Reconstruct Knapsack	Recover selected items
484	Edit Distance Alignment	Trace insert/delete/substitute
485	Matrix Chain Parentheses	Rebuild parenthesization
486	Coin Change Reconstruction	Backtrack last used coin
487	Path Reconstruction DP	Trace minimal route
488	Sequence Reconstruction	Rebuild from states
489	Multi-Choice Reconstruction	Combine best subpaths
490	Traceback Visualization	Visual DP backtrack tool

50. Meta-DP and Optimization Templates

#	Algorithm	Note
491	State Compression Template	Represent subsets compactly
492	Transition Optimization Template	Precompute transitions
493	Space Optimization Template	Rolling arrays
494	Multi-Dimensional DP Template	Nested loops version
495	Decision Monotonicity	Optimization hint
496	Monge Array Optimization	Matrix property leverage
497	Divide & Conquer Template	Half-split recursion
498	Rerooting Template	Generalized tree DP
499	Iterative DP Pattern	Bottom-up unrolling
500	Memoization Template	Recursive caching skeleton

Chapter 6. Mathematics for Algorithms

51. Number Theory (GCD, Modular Arithmetic, CRT)

#	Algorithm	Note
501	Euclidean Algorithm	Compute gcd(a, b)
502	Extended Euclidean Algorithm	Solve ax + by = gcd(a, b)
503	Modular Addition	Add under modulo M
504	Modular Multiplication	Multiply under modulo M
505	Modular Exponentiation	Fast power mod M
506	Modular Inverse	Compute a ^{1} mod M
507	Chinese Remainder Theorem	Combine modular systems
508	Binary GCD (Stein's Algorithm)	Bitwise gcd
509	Modular Reduction	Normalize residues
510	Modular Linear Equation Solver	Solve ax b (mod m)

52. Primality and Factorization (Miller–Rabin, Pollard Rho)

#	Algorithm	Note
511	Trial Division	Simple prime test
512	Sieve of Eratosthenes	Generate primes up to n
513	Sieve of Atkin	Faster sieve variant
514	Miller–Rabin Primality Test	Probabilistic primality
515	Fermat Primality Test	Modular power check
516	Pollard's Rho	Randomized factorization
517	Pollard's p−1 Method	Factor using smoothness
518	Wheel Factorization	Skip known composites
519	AKS Primality Test	Deterministic polynomial test
520	Segmented Sieve	Prime generation for large n

53. Combinatorics (Permutations, Combinations, Subsets)

#	Algorithm	Note
521	Factorial Precomputation	Build n! table
522	nCr Computation	Use Pascal's or factorials
523	Pascal's Triangle	Binomial coefficients
524	Multiset Combination	Repetition allowed

#	Algorithm	Note
525	Permutation Generation	Lexicographic order
526	Next Permutation	STL-style increment
527	Subset Generation	Bitmask or recursion
528	Gray Code Generation	Single-bit flips
529	Catalan Number DP	Count valid parentheses
530	Stirling Numbers	Partition counting

54. Probability and Randomized Algorithms

#	Algorithm	Note
531	Monte Carlo Simulation	Approximate via randomness
532	Las Vegas Algorithm	Always correct, variable time
533	Reservoir Sampling	Uniform sampling from stream
534	Randomized QuickSort	Expected O(n log n)
535	Randomized QuickSelect	Random pivot
536	Birthday Paradox Simulation	Probability collision
537	Random Hashing	Reduce collision chance
538	Random Walk Simulation	State transitions
539	Coupon Collector Estimation	Expected trials
540	Markov Chain Simulation	Transition matrix sampling

55. Sieve Methods and Modular Math

#	Algorithm	Note
541	Sieve of Eratosthenes	Base prime sieve
542	Linear Sieve	O(n) sieve variant
543	Segmented Sieve	Range prime generation
544	SPF (Smallest Prime Factor) Table	Factorization via sieve
545	Möbius Function Sieve	Multiplicative function calc
546	Euler's Totient Sieve	Compute (n) for all n
547	Divisor Count Sieve	Count divisors efficiently
548	Modular Precomputation	Store inverses, factorials
549	Fermat Little Theorem	a^(p−1) 1 mod p
550	Wilson's Theorem	Prime test via factorial mod p

56. Linear Algebra (Gaussian Elimination, LU, SVD)

#	Algorithm	Note
551	Gaussian Elimination	Solve Ax = b
552	Gauss-Jordan Elimination	Reduced row echelon
553	LU Decomposition	Factor A into L · U
554	Cholesky Decomposition	A = L · L for SPD
555	QR Decomposition	Orthogonal factorization
556	Matrix Inversion (Gauss-Jordan)	Find A ^{1}
557	Determinant by Elimination	Product of pivots
558	Rank of Matrix	Count non-zero rows
559	Eigenvalue Power Method	Approximate dominant eigenvalue
560	Singular Value Decomposition	A = UΣV

57. FFT and NTT (Fast Transforms)

#	Algorithm	Note
561	Discrete Fourier Transform (DFT)	O(n^2) baseline
562	Fast Fourier Transform (FFT)	O(n log n) convolution
563	Cooley–Tukey FFT	Recursive divide and conquer
564	Iterative FFT	In-place bit reversal
565	Inverse FFT	Recover time-domain
566	Convolution via FFT	Polynomial multiplication
567	Number Theoretic Transform (NTT)	Modulo prime FFT
568	Inverse NTT	Modular inverse transform
569	Bluestein's Algorithm	FFT of arbitrary size
570	FFT-Based Multiplication	Big integer product

58. Numerical Methods (Newton, Simpson, Runge–Kutta)

#	Algorithm	Note
571	Newton–Raphson	Root finding via tangent
572	Bisection Method	Interval halving
573	Secant Method	Approximate derivative
574	Fixed-Point Iteration	x = f(x) convergence
575	Gaussian Quadrature	Weighted integration
576	Simpson's Rule	Piecewise quadratic integral

#	Algorithm	Note
577	Trapezoidal Rule	Linear interpolation integral
578	Runge–Kutta (RK4)	ODE solver
579	Euler's Method	Step-by-step ODE
580	Gradient Descent (1D)	Numerical optimization

59. Mathematical Optimization (Simplex, Gradient, Convex)

#	Algorithm	Note
581	Simplex Method	Linear programming solver
582	Dual Simplex Method	Solve dual constraints
583	Interior-Point Method	Convex optimization
584	Gradient Descent	Unconstrained optimization
585	Stochastic Gradient Descent	Sample-based updates
586	Newton's Method (Multivariate)	Quadratic convergence
587	Conjugate Gradient	Solve SPD systems
588	Lagrange Multipliers	Constrained optimization
589	KKT Conditions Solver	Convex constraint handling
590	Coordinate Descent	Sequential variable updates

60. Algebraic Tricks and Transform Techniques

#	Algorithm	Note
591	Polynomial Multiplication (FFT)	Fast convolution
592	Polynomial Inversion	Newton iteration
593	Polynomial Derivative	Term-wise multiply by index
594	Polynomial Integration	Divide by index+1
595	Formal Power Series Composition	Substitute series
596	Exponentiation by Squaring	Fast powering
597	Modular Exponentiation	Fast power mod M
598	Fast Walsh–Hadamard Transform	XOR convolution
599	Zeta Transform	Subset summation
600	Möbius Inversion	Recover original from sums

Chapter 7. Strings and Text Algorithms

61. String Matching (KMP, Z, Rabin–Karp, Boyer–Moore)

#	Algorithm	Note
601	Naive String Matching	Compare every position
602	Knuth–Morris–Pratt (KMP)	Prefix function skipping
603	Z-Algorithm	Match using Z-values
604	Rabin–Karp	Rolling hash comparison
605	Boyer–Moore	Backward skip based on mismatch
606	Boyer–Moore–Horspool	Simplified shift table
607	Sunday Algorithm	Last-character shift
608	Finite Automaton Matching	DFA-based matching
609	Bitap Algorithm	Bitmask approximate matching
610	Two-Way Algorithm	Optimal linear matching

62. Multi-Pattern Search (Aho–Corasick)

#	Algorithm	Note
611	Aho–Corasick Automaton	Trie + failure links
612	Trie Construction	Prefix tree build
613	Failure Link Computation	BFS for transitions
614	Output Link Management	Handle overlapping patterns
615	Multi-Pattern Search	Find all keywords
616	Dictionary Matching	Find multiple substrings
617	Dynamic Aho–Corasick	Add/remove patterns
618	Parallel AC Search	Multi-threaded traversal
619	Compressed AC Automaton	Memory-optimized
620	Extended AC with Wildcards	Flexible matching

63. Suffix Structures (Suffix Array, Suffix Tree, LCP)

#	Algorithm	Note
621	Suffix Array (Naive)	Sort all suffixes
622	Suffix Array (Doubling)	O(n log n) rank-based
623	Kasai's LCP Algorithm	Longest common prefix
624	Suffix Tree (Ukkonen)	Linear-time online

#	Algorithm	Note
625	Suffix Automaton	Minimal DFA of substrings
626	SA-IS Algorithm	O(n) suffix array
627	LCP RMQ Query	Range minimum for substring
628	Generalized Suffix Array	Multiple strings
629	Enhanced Suffix Array	Combine SA + LCP
630	Sparse Suffix Tree	Space-efficient variant

64. Palindromes and Periodicity (Manacher)

#	Algorithm	Note
631	Naive Palindrome Check	Expand around center
632	Manacher's Algorithm	O(n) longest palindrome
633	Longest Palindromic Substring	Center expansion
634	Palindrome DP Table	Substring boolean matrix
635	Palindromic Tree (Eertree)	Track distinct palindromes
636	Prefix Function Periodicity	Detect repetition patterns
637	Z-Function Periodicity	Identify periodic suffix
638	KMP Prefix Period Check	Shortest repeating unit
639	Lyndon Factorization	Decompose string into Lyndon words
640	Minimal Rotation (Booth's Algorithm)	Lexicographically minimal shift

65. Edit Distance and Alignment

#	Algorithm	Note
641	Levenshtein Distance	Insert/delete/replace cost
642	Damerau–Levenshtein	Swap included
643	Hamming Distance	Count differing bits
644	Needleman–Wunsch	Global alignment
645	Smith–Waterman	Local alignment
646	Hirschberg's Algorithm	Memory-optimized alignment
647	Edit Script Reconstruction	Backtrack operations
648	Affine Gap Penalty DP	Varying gap cost
649	Myers Bit-Vector Algorithm	Fast edit distance
650	Longest Common Subsequence	Alignment by inclusion

66. Compression (Huffman, Arithmetic, LZ77, BWT)

#	Algorithm	Note
651	Huffman Coding	Optimal prefix tree
652	Canonical Huffman	Deterministic ordering
653	Arithmetic Coding	Interval probability coding
654	Shannon–Fano Coding	Early prefix method
655	Run-Length Encoding (RLE)	Repeat compression
656	LZ77	Sliding-window match
657	LZ78	Dictionary building
658	LZW	Variant used in GIF
659	Burrows–Wheeler Transform	Block reordering
660	Move-to-Front Encoding	Locality boosting transform

67. Cryptographic Hashes and Checksums

#	Algorithm	Note
661	Rolling Hash	Polynomial mod-based
662	CRC32	Cyclic redundancy check
663	Adler-32	Lightweight checksum
664	MD5	Legacy cryptographic hash
665	SHA-1	Deprecated hash function
666	SHA-256	Secure hash standard
667	SHA-3 (Keccak)	Sponge construction
668	HMAC	Keyed message authentication
669	Merkle Tree	Hierarchical hashing
670	Hash Collision Detection	Birthday bound simulation

68. Approximate and Streaming Matching

#	Algorithm	Note
671	K-Approximate Matching	Allow k mismatches
672	Bitap Algorithm	Bitwise dynamic programming
673	Landau–Vishkin Algorithm	Edit distance k
674	Filtering Algorithm	Fast approximate search
675	Wu–Manber	Multi-pattern approximate search
676	Streaming KMP	Online prefix updates

#	Algorithm	Note
677	Rolling Hash Sketch	Sliding window hashing
678	Sketch-based Similarity	MinHash / LSH variants
679	Weighted Edit Distance	Weighted operations
680	Online Levenshtein	Dynamic stream update

69. Bioinformatics Alignment (Needleman–Wunsch, Smith–Waterman)

#	Algorithm	Note
681	Needleman–Wunsch	Global sequence alignment
682	Smith–Waterman	Local alignment
683	Gotoh Algorithm	Affine gap penalties
684	Hirschberg Alignment	Linear-space alignment
685	Multiple Sequence Alignment (MSA)	Progressive methods
686	Profile Alignment	Align sequence to profile
687	Hidden Markov Model Alignment	Probabilistic alignment
688	BLAST	Heuristic local search
689	FASTA	Word-based alignment
690	Pairwise DP Alignment	General DP framework

70. Text Indexing and Search Structures

#	Algorithm	Note
691	Inverted Index Build	Word-to-document mapping
692	Positional Index	Store word positions
693	TF-IDF Weighting	Importance scoring
694	BM25 Ranking	Modern ranking formula
695	Trie Index	Prefix search structure
696	Suffix Array Index	Substring search
697	Compressed Suffix Array	Space-optimized
698	FM-Index	BWT-based compressed index
699	DAWG (Directed Acyclic Word Graph)	Shared suffix graph
700	Wavelet Tree for Text	Rank/select on sequences

Chapter 8. Geometry, Graphics, and Spatial Algorithms

71. Convex Hull (Graham, Andrew, Chan)

#	Algorithm	Note
701	Gift Wrapping (Jarvis March)	Wrap hull one point at a time
702	Graham Scan	Sort by angle, maintain stack
703	Andrew's Monotone Chain	Sort by x, upper + lower hull
704	Chan's Algorithm	Output-sensitive O(n log h)
705	QuickHull	Divide-and-conquer hull
706	Incremental Convex Hull	Add points one by one
707	Divide & Conquer Hull	Merge two partial hulls
708	3D Convex Hull	Extend to 3D geometry
709	Dynamic Convex Hull	Maintain hull with inserts
710	Rotating Calipers	Compute diameter, width, antipodal pairs

72. Closest Pair and Segment Intersection

#	Algorithm	Note
711	Closest Pair (Divide & Conquer)	Split, merge minimal distance
712	Closest Pair (Sweep Line)	Maintain active window
713	Brute Force Closest Pair	Check all $O(n^2)$ pairs
714	Bentley–Ottmann	Find all line intersections
715	Segment Intersection Test	Cross product orientation
716	Line Sweep for Segments	Event-based intersection
717	Intersection via Orientation	CCW test
718	Circle Intersection	Geometry of two circles
719	Polygon Intersection	Clip overlapping polygons
720	Nearest Neighbor Pair	Combine KD-tree + search

73. Line Sweep and Plane Sweep Algorithms

#	Algorithm	Note
721	Sweep Line for Events	Process sorted events
722	Interval Scheduling	Select non-overlapping intervals
723	Rectangle Union Area	Sweep edges to count area
724	Segment Intersection (Bentley–Ottmann)	Detect all crossings

#	Algorithm	Note
725	Skyline Problem	Merge height profiles
726	Closest Pair Sweep	Maintain active set
727	Circle Arrangement	Sweep and count regions
728	Sweep for Overlapping Rectangles	Detect collisions
729	Range Counting	Count points in rectangle
730	Plane Sweep for Triangles	Polygon overlay computation

74. Delaunay and Voronoi Diagrams

#	Algorithm	Note
731	Delaunay Triangulation (Incremental)	Add points, maintain Delaunay
732	Delaunay (Divide & Conquer)	Merge triangulations
733	Delaunay (Fortune's Sweep)	O(n log n) construction
734	Voronoi Diagram (Fortune's)	Sweep line beachline
735	Incremental Voronoi	Update on insertion
736	Bowyer–Watson	Empty circle criterion
737	Duality Transform	Convert between Voronoi/Delaunay
738	Power Diagram	Weighted Voronoi
739	Lloyd's Relaxation	Smooth Voronoi cells
740	Voronoi Nearest Neighbor	Region-based lookup

75. Point in Polygon and Polygon Triangulation

#	Algorithm	Note
741	Ray Casting	Count edge crossings
742	Winding Number	Angle sum method
743	Convex Polygon Point Test	Orientation checks
744	Ear Clipping Triangulation	Remove ears iteratively
745	Monotone Polygon Triangulation	Sweep line triangulation
746	Delaunay Triangulation	Optimal triangle quality
747	Convex Decomposition	Split into convex parts
748	Polygon Area (Shoelace Formula)	Signed area computation
749	Minkowski Sum	Add shapes geometrically
750	Polygon Intersection (Weiler–Atherton)	Clip overlapping shapes

76. Spatial Data Structures (KD, R-tree)

#	Algorithm	Note
751	KD-Tree Build	Recursive median split
752	KD-Tree Search	Axis-aligned query
753	Range Search KD-Tree	Orthogonal query
754	Nearest Neighbor KD-Tree	Closest point search
755	R-Tree Build	Bounding box hierarchy
756	R*-Tree	Optimized split strategy
757	Quad Tree	Spatial decomposition
758	Octree	3D spatial decomposition
759	BSP Tree (Binary Space Partition)	Split by planes
760	Morton Order (Z-Curve)	Spatial locality index

77. Rasterization and Scanline Techniques

#	Algorithm	Note
761	Bresenham's Line Algorithm	Efficient integer drawing
762	Midpoint Circle Algorithm	Circle rasterization
763	Scanline Fill	Polygon interior fill
764	Edge Table Fill	Sort edges by y
765	Z-Buffer Algorithm	Hidden surface removal
766	Painter's Algorithm	Sort by depth
767	Gouraud Shading	Vertex interpolation shading
768	Phong Shading	Normal interpolation
769	Anti-Aliasing (Supersampling)	Smooth jagged edges
770	Scanline Polygon Clipping	Efficient clipping

78. Computer Vision (Canny, Hough, SIFT)

#	Algorithm	Note
771	Canny Edge Detector	Gradient + hysteresis
772	Sobel Operator	Gradient magnitude filter
773	Hough Transform (Lines)	Accumulator for line detection
774	Hough Transform (Circles)	Radius-based accumulator
775	Harris Corner Detector	Eigenvalue-based corners
776	FAST Corner Detector	Intensity circle test

#	Algorithm	Note
777	SIFT (Scale-Invariant Feature Transform)	Keypoint detection
778	SURF (Speeded-Up Robust Features)	Faster descriptor
779	ORB (Oriented FAST + BRIEF)	Binary robust feature
780	RANSAC	Robust model fitting

79. Pathfinding in Space (A*, RRT, PRM)

#	Algorithm	Note
781	A* Search	Heuristic pathfinding
782	Dijkstra for Grid	Weighted shortest path
783	Theta*	Any-angle pathfinding
784	Jump Point Search	Grid acceleration
785	RRT (Rapidly-Exploring Random Tree)	Random sampling tree
786	RRT*	Optimal variant with rewiring
787	PRM (Probabilistic Roadmap)	Graph sampling planner
788	Visibility Graph	Connect visible vertices
789	Potential Field Pathfinding	Gradient-based navigation
790	Bug Algorithms	Simple obstacle avoidance

80. Computational Geometry Variants and Applications

#	Algorithm	Note
791	Convex Polygon Intersection	Clip convex sets
792	Minkowski Sum	Shape convolution
793	Rotating Calipers	Closest/farthest pair
794	Half-Plane Intersection	Feasible region
795	Line Arrangement	Count regions
796	Point Location (Trapezoidal Map)	Query region lookup
797	Voronoi Nearest Facility	Region query
798	Delaunay Mesh Generation	Triangulation refinement
799	Smallest Enclosing Circle	Welzl's algorithm
800	Collision Detection (SAT)	Separating axis theorem

Chapter 9. Systems, Databases, and Distributed Algorithms

81. Concurrency Control (2PL, MVCC, OCC)

#	Algorithm	Note
801	Two-Phase Locking (2PL)	Acquire-then-release locks
802	Strict 2PL	Hold locks until commit
803	Conservative 2PL	Prevent deadlocks via prelock
804	Timestamp Ordering	Schedule by timestamps
805	Multiversion Concurrency Control (MVCC)	Snapshot isolation
806	Optimistic Concurrency Control (OCC)	Validate at commit
807	Serializable Snapshot Isolation	Merge read/write sets
808	Lock-Free Algorithm	Atomic CAS updates
809	Wait-Die / Wound-Wait	Deadlock prevention policies
810	Deadlock Detection (Wait-for Graph)	Cycle detection in waits

82. Logging, Recovery, and Commit Protocols

#	Algorithm	Note
811	Write-Ahead Logging (WAL)	Log before commit
812	ARIES Recovery	Re-do/undo with LSNs
813	Shadow Paging	Copy-on-write persistence
814	Two-Phase Commit (2PC)	Coordinator-driven commit
815	Three-Phase Commit (3PC)	Non-blocking variant
816	Checkpointing	Save state for recovery
817	Undo Logging	Rollback uncommitted
818	Redo Logging	Reapply committed
819	Quorum Commit	Majority agreement
820	Consensus Commit	Combine 2PC + Paxos

83. Scheduling (Round Robin, EDF, Rate-Monotonic)

#	Algorithm	Note
821	First-Come First-Served (FCFS)	Sequential job order
822	Shortest Job First (SJF)	Optimal average wait
823	Round Robin (RR)	Time-slice fairness
824	Priority Scheduling	Weighted selection

#	Algorithm	Note
825	Multilevel Queue	Tiered priority queues
826	Earliest Deadline First (EDF)	Real-time optimal
827	Rate Monotonic Scheduling (RMS)	Fixed periodic priority
828	Lottery Scheduling	Probabilistic fairness
829	Multilevel Feedback Queue	Adaptive behavior
830	Fair Queuing (FQ)	Flow-based proportional sharing

84. Caching and Replacement (LRU, LFU, CLOCK)

#	Algorithm	Note
831	LRU (Least Recently Used)	Evict oldest used
832	LFU (Least Frequently Used)	Evict lowest frequency
833	FIFO Cache	Simple queue eviction
834	CLOCK Algorithm	Approximate LRU
835	ARC (Adaptive Replacement Cache)	Mix of recency + frequency
836	Two-Queue (2Q)	Separate recent/frequent
837	LIRS (Low Inter-reference Recency Set)	Predict reuse distance
838	TinyLFU	Frequency sketch admission
839	Random Replacement	Simple stochastic policy
840	Belady's Optimal	Evict farthest future use

85. Networking (Routing, Congestion Control)

#	Algorithm	Note
841	Dijkstra's Routing	Shortest path routing
842	Bellman–Ford Routing	Distance-vector routing
843	Link-State Routing (OSPF)	Global view routing
844	Distance-Vector Routing (RIP)	Local neighbor updates
845	Path Vector (BGP)	Route advertisement
846	Flooding	Broadcast to all nodes
847	Spanning Tree Protocol	Loop-free topology
848	Congestion Control (AIMD)	TCP window control
849	Random Early Detection (RED)	Queue preemptive drop
850	ECN (Explicit Congestion Notification)	Mark packets early

86. Distributed Consensus (Paxos, Raft, PBFT)

#	Algorithm	Note
851	Basic Paxos	Majority consensus
852	Multi-Paxos	Sequence of agreements
853	Raft	Log replication + leader election
854	Viewstamped Replication	Alternative consensus design
855	PBFT (Practical Byzantine Fault Tolerance)	Byzantine safety
856	Zab (Zookeeper Atomic Broadcast)	Broadcast + ordering
857	EPaxos	Leaderless fast path
858	VRR (Virtual Ring Replication)	Log around ring
859	Two-Phase Commit with Consensus	Transactional commit
860	Chain Replication	Ordered state replication

87. Load Balancing and Rate Limiting

#	Algorithm	Note
861	Round Robin Load Balancing	Sequential distribution
862	Weighted Round Robin	Proportional to weight
863	Least Connections	Pick least loaded node
864	Consistent Hashing	Map requests stably
865	Power of Two Choices	Sample and choose lesser load
866	Random Load Balancing	Simple uniform random
867	Token Bucket	Rate-based limiter
868	Leaky Bucket	Steady flow shaping
869	Sliding Window Counter	Rolling time window
870	Fixed Window Counter	Resettable counter limiter

88. Search and Indexing (Inverted, BM25, WAND)

#	Algorithm	Note
871	Inverted Index Construction	Word → document list
872	Positional Index Build	Store term positions
873	TF-IDF Scoring	Term frequency weighting
874	BM25 Ranking	Modern scoring model
875	Boolean Retrieval	Logical AND/OR/NOT
876	WAND Algorithm	Efficient top-k retrieval

#	Algorithm	Note
877	Block-Max WAND (BMW)	Early skipping optimization
878	Impact-Ordered Indexing	Sort by contribution
879	Tiered Indexing	Prioritize high-score docs
880	DAAT vs SAAT Evaluation	Document vs score-at-a-time

89. Compression and Encoding in Systems

#	Algorithm	Note
881	Run-Length Encoding (RLE)	Simple repetition encoding
882	Huffman Coding	Optimal variable-length code
883	Arithmetic Coding	Fractional interval coding
884	Delta Encoding	Store differences
885	Variable Byte Encoding	Compact integers
886	Elias Gamma Coding	Prefix integer encoding
887	Rice Coding	Unary + remainder scheme
888	Snappy	Fast block compression
889	Zstandard (Zstd)	Modern adaptive codec
890	LZ4	High-speed dictionary compressor

90. Fault Tolerance and Replication

#	Algorithm	Note
891	Primary–Backup Replication	One leader, one standby
892	Quorum Replication	Majority write/read rule
893	Chain Replication	Ordered consistency
894	Gossip Protocol	Epidemic state exchange
895	Anti-Entropy Repair	Periodic reconciliation
896	Erasure Coding	Redundant data blocks
897	Checksum Verification	Detect corruption
898	Heartbeat Monitoring	Liveness detection
899	Leader Election (Bully)	Highest ID wins
900	Leader Election (Ring)	Token-based rotation

Chapter 10. AI, ML, and Optimization

91. Classical ML (k-means, Naive Bayes, SVM, Decision Trees)

#	Algorithm	Note
901	k-Means Clustering	Partition by centroid iteration
902	k-Medoids (PAM)	Cluster by exemplars
903	Gaussian Mixture Model (EM)	Soft probabilistic clustering
904	Naive Bayes Classifier	Probabilistic feature independence
905	Logistic Regression	Sigmoid linear classifier
906	Perceptron	Online linear separator
907	Decision Tree (CART)	Recursive partition by impurity
908	ID3 Algorithm	Information gain splitting
909	k-Nearest Neighbors (kNN)	Distance-based classification
910	Linear Discriminant Analysis (LDA)	Projection for separation

92. Ensemble Methods (Bagging, Boosting, Random Forests)

#	Algorithm	Note
911	Bagging	Bootstrap aggregation
912	Random Forest	Ensemble of decision trees
913	AdaBoost	Weighted error correction
914	Gradient Boosting	Sequential residual fitting
915	XGBoost	Optimized gradient boosting
916	LightGBM	Histogram-based leaf growth
917	CatBoost	Ordered boosting for categoricals
918	Stacking	Meta-model ensemble
919	Voting Classifier	Majority aggregation
920	Snapshot Ensemble	Averaged checkpoints

93. Gradient Methods (SGD, Adam, RMSProp)

#	Algorithm	Note
921	Gradient Descent	Batch full-gradient step
922	Stochastic Gradient Descent (SGD)	Sample-wise updates
923	Mini-Batch SGD	Tradeoff speed and variance
924	Momentum	Add velocity to descent

#	Algorithm	Note
925	Nesterov Accelerated Gradient	Lookahead correction
926	AdaGrad	Adaptive per-parameter rate
927	RMSProp	Exponential moving average
928	Adam	Momentum + adaptive rate
929	AdamW	Decoupled weight decay
930	L-BFGS	Limited-memory quasi-Newton

94. Deep Learning (Backpropagation, Dropout, Normalization)

#	Algorithm	Note
931	Backpropagation	Gradient chain rule
932	Xavier/He Initialization	Scaled variance init
933	Dropout	Random neuron deactivation
934	Batch Normalization	Normalize per batch
935	Layer Normalization	Normalize per feature
936	Gradient Clipping	Prevent explosion
937	Early Stopping	Prevent overfitting
938	Weight Decay	Regularization via penalty
939	Learning Rate Scheduling	Dynamic LR adjustment
940	Residual Connections	Skip layer improvement

95. Sequence Models (Viterbi, Beam Search, CTC)

#	Algorithm	Note
941	Hidden Markov Model (Forward–Backward)	Probabilistic sequence model
942	Viterbi Algorithm	Most probable path
943	Baum–Welch	EM training for HMMs
944	Beam Search	Top-k path exploration
945	Greedy Decoding	Fast approximate decoding
946	Connectionist Temporal Classification (CTC)	Unaligned sequence training
947	Attention Mechanism	Weighted context aggregation
948	Transformer Decoder	Self-attention stack
949	Seq2Seq with Attention	Encoder-decoder framework
950	Pointer Network	Output index selection

96. Metaheuristics (GA, SA, PSO, ACO)

#	Algorithm	Note
951	Genetic Algorithm (GA)	Evolutionary optimization
952	Simulated Annealing (SA)	Temperature-controlled search
953	Tabu Search	Memory of forbidden moves
954	Particle Swarm Optimization (PSO)	Velocity-based search
955	Ant Colony Optimization (ACO)	Pheromone-guided path
956	Differential Evolution (DE)	Vector-based mutation
957	Harmony Search	Music-inspired improvisation
958	Firefly Algorithm	Brightness-attraction movement
959	Bee Colony Optimization	Explore-exploit via scouts
960	Hill Climbing	Local incremental improvement

97. Reinforcement Learning (Q-learning, Policy Gradients)

#	Algorithm	Note
961	Monte Carlo Control	Average returns
962	Temporal Difference (TD) Learning	Bootstrap updates
963	SARSA	On-policy TD learning
964	Q-Learning	Off-policy TD learning
965	Double Q-Learning	Reduce overestimation
966	Deep Q-Network (DQN)	Neural Q approximator
967	REINFORCE	Policy gradient by sampling
968	Actor–Critic	Value-guided policy update
969	PPO (Proximal Policy Optimization)	Clipped surrogate objective
970	DDPG / SAC	Continuous action RL

98. Approximation and Online Algorithms

#	Algorithm	Note
971	Greedy Set Cover	ln(n)-approximation
972	Vertex Cover Approximation	Double-matching heuristic
973	Traveling Salesman Approximation	MST-based 2-approx
974	k-Center Approximation	Farthest-point heuristic
975	Online Paging (LRU)	Competitive analysis
976	Online Matching (Ranking)	Adversarial input resilience

#	Algorithm	Note
977	Online Knapsack	Ratio-based acceptance
978	Competitive Ratio Evaluation	Bound worst-case performance
979	PTAS / FPTAS Schemes	Polynomial approximation
980	Primal–Dual Method	Approximate combinatorial optimization

99. Fairness, Causal Inference, and Robust Optimization

#	Algorithm	Note
981	Reweighting for Fairness	Adjust sample weights
982	Demographic Parity Constraint	Equalize positive rates
983	Equalized Odds	Align error rates
984	Adversarial Debiasing	Learn fair representations
985	Causal DAG Discovery	Graphical causal inference
986	Propensity Score Matching	Estimate treatment effect
987	Instrumental Variable Estimation	Handle confounders
988	Robust Optimization	Worst-case aware optimization
989	Distributionally Robust Optimization	Minimax over uncertainty sets
990	Counterfactual Fairness	Simulate do-interventions

100. AI Planning, Search, and Learning Systems

#	Algorithm	Note
991	Breadth-First Search (BFS)	Uninformed search
992	Depth-First Search (DFS)	Backtracking search
993	A* Search	Heuristic guided
994	Iterative Deepening A* (IDA*)	Memory-bounded heuristic
995	Uniform Cost Search	Expand by path cost
996	Monte Carlo Tree Search (MCTS)	Exploration vs exploitation
997	Minimax	Game tree evaluation
998	Alpha–Beta Pruning	Prune unneeded branches
999	STRIPS Planning	Action-based state transition
1000	Hierarchical Task Network (HTN)	Structured AI planning