University of Calicut

B.Sc. Mathematics Honours — CUFYUGP 2024

Semester V Elective — MAT5EJ302(1)

Data Structures and Algorithms

A Beginner-Friendly Study Guide

Build intuition. Understand deeply. Prepare for exams.

Based on:

S. Dasgupta, C. Papadimitriou, and U. Vazirani,
Algorithms, McGraw-Hill, 2006

Academic Year 2026–27

Preface

This book is a study guide for **MAT5EJ302(1) — Data Structures and Algorithms**, an elective course offered to B.Sc. Mathematics Honours students at colleges affiliated with the University of Calicut under the CUFYUGP 2024 regulations.

Who is this guide for? You are a mathematics student who is new to computing. You may have never written a program before. This guide assumes no prior programming experience. We explain every computing idea from scratch, drawing on analogies from everyday mathematics.

How to read this guide. Each chapter covers one module of the syllabus. Within each chapter, the sections follow this flow:

1. *Motivation* — why do we need this idea?
2. **Intuition box** (green) — the big idea in plain English.
3. *Formal definition or theorem* — the precise mathematical statement.
4. **Worked Example box** (orange) — a fully traced example, step by step.
5. *Pseudocode* — the algorithm written in plain, structured English.
6. **Python Code box** (gray) — the actual computer program.
7. **Exam Tip box** (red) — what the university exam tests on this topic.
8. **Key Takeaway box** (purple) — the one sentence to remember.

Not every section has all of these. Short topics may only have a definition and an example.

About the Python code. This course requires students to implement algorithms in Python. Rather than keeping the code in a separate module, we have integrated the Python implementations directly into the relevant sections. If you are completely new to Python, read the short *"Reading Python"* primer in Section 1.2 before proceeding to the first code box.

Exam pattern.

Component	Marks	Passing minimum
Internal assessment	30	—
External examination	70	15 per module (Modules I–IV)

The external exam covers **Modules I through IV** only. Module V (Python implementation) is assessed internally. Each module carries a minimum of 15 marks in the external exam, so every module deserves equal attention.

Notation. We write $\mathcal{O}(f(n))$ for "of order $f(n)$" (Big-O). We use $|V|$ and $|E|$ for the number of vertices and edges in a graph. The symbol log always means logarithm base 2 unless stated otherwise.

Contents

Chapter 1

Introduction to Algorithms

Module I · 12 Hours · Sections 0.2, 0.3, 1.1, 1.2, 1.3 of Dasgupta et al.

What you will learn in this chapter.

- What an algorithm is, and why its efficiency matters.
- How to measure and compare the speed of algorithms using Big-O notation.
- Three ways to compute Fibonacci numbers, with very different speeds.
- How computers perform arithmetic on large numbers.
- Modular arithmetic and why it is the foundation of cryptography.
- Euclid's celebrated algorithm for finding the greatest common divisor.
- Two methods for testing whether a number is prime.

1.1 What Is an Algorithm?

A recipe tells you how to bake a cake. It lists the ingredients, and then gives step-by-step instructions. If you follow the steps correctly, you get a cake every time. An algorithm is exactly the same idea, but for a mathematical or computational problem.

Definition

An **algorithm** is a finite, step-by-step procedure that takes some *input*, performs a sequence of well-defined operations, and produces the correct *output* for every valid input.

Two properties matter above all others:

1. **Correctness.** The algorithm must produce the right answer. Not just sometimes — it must work for *every* valid input.
2. **Efficiency.** The algorithm must finish in a reasonable amount of time (and use a reasonable amount of memory).

Intuition

Correctness is the bare minimum. But an algorithm that takes a million years to run is not useful, even if it would eventually give the right answer. Efficiency is what separates a practical algorithm from a theoretical curiosity.

Throughout this course, we study algorithms that are both *correct* and *efficient*. Our main tool for measuring efficiency is **Big-O notation**, introduced in Section 1.4.

1.2 Reading Python: A Quick Primer

This course requires you to implement algorithms in the Python programming language. You do not need prior programming experience. This short section gives you just enough knowledge to read and understand the Python code in the boxes throughout this guide. You will learn to *write* programs during the practical sessions with your instructor.

Variables

A **variable** is a named storage location. When you write `x = 5`, you are telling Python: "store the value 5, and call it `x`." You can later use `x` in calculations, and even change its value.

Functions

A **function** is a named block of code that performs one task. In Python, functions are defined with the keyword `def`:

Python Code

```
def square(n):          # "square" takes one input called n
    return n * n        # compute n*n and send it back to the
        caller
```

The word `return` sends the result back. To use the function: `square(4)` gives `16`.

Conditionals (if/else)

```
if n == 0:
    return 0            # this runs only when n equals 0
else:
    return n * n        # this runs otherwise
```

Python uses indentation (spaces) to show which lines belong together.

Loops

A `for` loop repeats code a fixed number of times. A `while` loop repeats as long as a condition is true.

```
for i in range(5):      # runs 5 times: i = 0, 1, 2, 3, 4
    print(i)

x = 10
while x > 0:            # runs as long as x is positive
    x = x - 3
```

Recursion

A **recursive** function is one that calls itself. Think of it like a Russian nesting doll: each doll contains a smaller version of itself. A recursive function must have a *base case* (the smallest doll, with no more nesting) to stop the process.

Lists and Dictionaries

A `list` stores a sequence of values: `a = [3, 1, 4, 1, 5]`. A `dict` (dictionary) stores key–value pairs: `memo = {3: 2, 4: 3}`. You look up values by key: `memo[3]` gives `2`.

Exam Tip

You will not be asked to write Python code in the external exam. However, the exam *will* ask you to trace algorithms step by step. The Python code in this guide will help you see the algorithm running concretely.

1.3 Computing Fibonacci Numbers

The **Fibonacci sequence** is defined as:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2.$$

The first few values are: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$

This seemingly simple sequence will reveal a deep lesson: the *same correct answer* can be computed in wildly different amounts of time, depending on how you design your algorithm.

1.3.1 Algorithm 1: Naive Recursion

The most natural approach mirrors the mathematical definition directly. To compute F_n, recursively compute F_{n-1} and F_{n-2}, then add them.

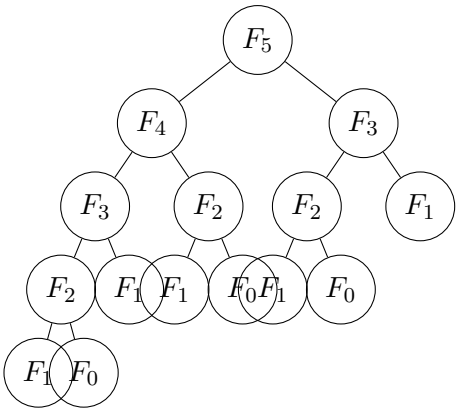
Algorithm: fib1(n)

```
function fib1(n)
    if n = 0: return 0
    if n = 1: return 1
    return fib1(n-1) + fib1(n-2)
```

Intuition

Imagine asking a friend: "What is F_5?" They say: "I need F_4 and F_3 first." So you ask two more friends. Each of *them* asks two more friends, and so on. The problem is that many of these friends are asked the *same question* repeatedly — a huge waste of effort.

To see how bad this is, look at the tree of recursive calls for F_5:



Notice that F_3 is computed *twice*, F_2 is computed *three times*, and F_1 is computed *five times*. For larger n, this explosion is catastrophic. The number of function calls grows like φ^n, where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio.

Definition

The running time of recursive Fibonacci is $\mathcal{O}(\varphi^n)$, which is **exponential**. For $n = 100$, this would require more operations than the number of atoms in the observable universe.

1.3.2 Algorithm 2: Memoisation (Top-Down Dynamic Programming)

The fix is simple: *remember* every answer you compute, so you never compute the same thing twice. This technique is called **memoisation** (from "memo" — a note you write to yourself).

Algorithm: fib2(n, memo)

```
function fib2(n, memo)
    if n in memo: return memo[n]
    if n = 0: return 0
    if n = 1: return 1
    memo[n] = fib2(n-1, memo) + fib2(n-2, memo)
    return memo[n]
```

Intuition

Now each friend keeps a personal notebook. The first time they are asked "What is F_k?", they compute and *write down* the answer. The next time anyone asks, they just read from the notebook. This means each F_k is computed exactly once, for a total of n computations.

The running time drops from $\mathcal{O}(\varphi^n)$ to $\mathcal{O}(n)$ — a dramatic improvement.

1.3.3 Algorithm 3: Iterative (Bottom-Up)

We can do even better in terms of *space*. Instead of recursion, we simply iterate from F_0 up to F_n, keeping only the two most recent values.

Algorithm: fib3(n)

```
function fib3(n)
    if n = 0: return 0
    a = 0, b = 1
    for i = 2 to n:
        a, b = b, a + b
    return b
```

Trace for $n = 6$

Step i	a (was F_{i-2})	b (was F_{i-1})	New b ($= F_i$)
Start	0	1	—
$i = 2$	1	$0 + 1 = 1$	1
$i = 3$	1	$1 + 1 = 2$	2
$i = 4$	2	$1 + 2 = 3$	3
$i = 5$	3	$2 + 3 = 5$	5
$i = 6$	5	$3 + 5 = 8$	8

Result: $F_6 = 8$. ✓

The iterative algorithm runs in $\mathcal{O}(n)$ time and uses only $\mathcal{O}(1)$ extra memory (just two variables a and b).

1.3.4 Python Implementation

Python Code

```python
def fib_recursive(n):
    if n == 0:                # base case
        return 0
    if n == 1:                # base case
        return 1
    return fib_recursive(n - 1) + fib_recursive(n - 2)  # recursive case

def fib_memo(n, memo=None):
    if memo is None:
        memo = {}             # start with an empty notebook
    if n in memo:
        return memo[n]        # already computed -- use saved answer
    if n == 0: return 0
    if n == 1: return 1
    memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)
```

```
    return memo[n]           # save before returning

def fib_iterative(n):
    if n == 0: return 0
    a, b = 0, 1              # a = F(0), b = F(1)
    for _ in range(2, n + 1):
        a, b = b, a + b      # shift: new b = old a + old b
    return b
```

Try it: Trace `fib_iterative(5)` by hand, following the loop iteration by iteration.

Warning

Do not call `fib_recursive(n)` for $n > 40$ on a computer. It will take minutes or hours. The iterative version computes F_{1000} in a tiny fraction of a second.

Exam Tip

A very common exam question: **"How many recursive calls does the naive Fibonacci algorithm make for F_n?"**
The answer: the number of calls $T(n)$ satisfies $T(n) = T(n-1) + T(n-2) + 1$, which gives $T(n) = \mathcal{O}(\varphi^n)$.
You may also be asked to **compare the three algorithms** in a table (time complexity + space complexity).

Key Takeaway

Memoisation converts exponential-time recursion into linear-time computation by storing and reusing previously computed answers.

1.4 Measuring Efficiency: Big-O Notation

When we say algorithm A is "faster" than algorithm B, we need a precise way to compare them. We cannot just run both on a computer and time them, because the result depends on the machine, the programming language, and other factors. We need a machine-independent measure of efficiency.

1.4.1 The Size of the Input

The running time of an algorithm depends on the *size* of its input. Sorting 10 numbers is faster than sorting a million. We express running time as a function of input size n.

For numerical algorithms, "size" means the number of *bits* needed to write down the input. A number N requires about $\log_2 N$ bits. For sorting and searching algorithms, "size" usually means the number of items.

1.4.2 Asymptotic Growth

We do not need an exact formula; we need to know how the running time *grows* as n increases. The key insight is: *for large n, the dominant term determines everything.*

Definition

We say $f(n) = \mathcal{O}(g(n))$ if there exist positive constants c and n_0 such that

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

We read this as "f is Big-O of g", or "f grows no faster than g".

Intuition

Big-O is like a speed limit. If your car has a speed limit of 100 km/h (Big-O of 100), it doesn't matter if it sometimes goes 80 or 95 — the limit tells you the *worst case*. Similarly, $\mathcal{O}(n^2)$ tells us the running time doesn't grow faster than a constant times n^2.
We always drop constant factors and lower-order terms. Why? Because for large n, they become negligible. For example: $3n^2 + 100n + 5 = \mathcal{O}(n^2)$.

1.4.3 Common Complexity Classes

The table below shows how different complexity classes compare. Assume your algorithm runs 10^8 operations per second.

Complexity	**Name**	$n = 10$	$n = 100$	$n = 1000$	$n = 10^6$
$\mathcal{O}(\log n)$	Logarithmic	3 ns	7 ns	10 ns	20 ns
$\mathcal{O}(n)$	Linear	0.1 μs	1 μs	10 μs	10 ms
$\mathcal{O}(n \log n)$	Linearithmic	0.3 μs	7 μs	100 μs	200 ms
$\mathcal{O}(n^2)$	Quadratic	1 μs	100 μs	100 ms	$> 10^5$ s
$\mathcal{O}(n^3)$	Cubic	10 μs	100 ms	hours	impossible
$\mathcal{O}(2^n)$	Exponential	1 μs	10^{13} yrs	∞	∞

Algorithms with polynomial complexity ($\mathcal{O}(n^k)$ for some constant k) are called **efficient**. Algorithms with exponential complexity are **inefficient** for large inputs.

Exam Tip

Know how to simplify Big-O expressions:

- $\mathcal{O}(5n^2 + 3n + 7) = \mathcal{O}(n^2)$ (keep the dominant term, drop constants).
- $\mathcal{O}(\log_3 n) = \mathcal{O}(\log n)$ (logarithm base doesn't matter in Big-O).
- If an algorithm has two nested loops each running n times, it is $\mathcal{O}(n^2)$.
- If an algorithm halves its input at each step, it is $\mathcal{O}(\log n)$.

Key Takeaway

$\mathcal{O}(g(n))$ captures the worst-case growth rate of an algorithm's running time, ignoring constant factors.

1.5 Algorithms with Numbers

Before studying clever algorithms, we should understand how computers store and compute with numbers. This sets the stage for modular arithmetic and cryptography.

1.5.1 Representing Numbers in Binary

Humans use base 10 (decimal): we have 10 digits, 0–9. Computers use base 2 (binary): only two digits, 0 and 1. Any positive integer N can be written in binary. The binary representation of N requires $\lfloor \log_2 N \rfloor + 1$ bits.

Binary representation

$13 = 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = (1101)_2$
The decimal number 13 needs 4 bits in binary. In general, N needs about $\log_2 N$ bits.

Why does this matter for algorithms? When we count the "size" of a number N as an input, we mean the number of bits, not the value. A 64-bit number can hold values up to $2^{64} \approx 1.8 \times 10^{19}$. An algorithm that is efficient in terms of the *value* of N might still be slow in terms of the *bit-length* of N.

1.5.2 Addition

To add two n-bit numbers, we go bit by bit from right to left, keeping a carry. This is exactly how you add in decimal, but in base 2.

The number of operations is proportional to n (the number of bits).

Addition of two n-bit numbers runs in $\mathcal{O}(n)$.

1.5.3 Multiplication

The grade-school method for multiplying two n-bit numbers: multiply the first number by each bit of the second, shift appropriately, and add all results. This produces n rows, each of length up to $2n$, requiring about n^2 bit operations.

Grade-school multiplication of two n-bit numbers runs in $\mathcal{O}(n^2)$.

In Chapter 2, we will see Karatsuba's algorithm, which reduces this to $\mathcal{O}(n^{1.585})$.

Key Takeaway

The "size" of a number for algorithmic purposes is its bit-length ($\approx \log N$), not its value.

1.6 Modular Arithmetic

1.6.1 The Clock Analogy

Think of a 12-hour clock. After 12 o'clock, we restart from 1. If it is 10 o'clock and we add 5 hours, we get 3 o'clock (not 15). This is *arithmetic modulo 12*: we always reduce

by multiples of 12 to stay within $\{0, 1, \ldots, 11\}$.

Definition

For a positive integer N (called the **modulus**), we say a and b are **congruent modulo** N, written $a \equiv b \pmod{N}$, if N divides $(a - b)$.
Equivalently, $a \bmod N$ is the remainder when a is divided by N.

Modular arithmetic examples

$$\begin{aligned} 17 \bmod 5 &= 2 \quad (\text{because } 17 = 3 \times 5 + 2) \\ 29 \equiv 1 \pmod{7} &\quad (\text{because } 29 - 1 = 28 = 4 \times 7) \\ -3 \equiv 9 \pmod{12} &\quad (\text{because } -3 + 12 = 9) \end{aligned}$$

1.6.2 Properties of Modular Arithmetic

Modular arithmetic behaves like ordinary arithmetic for addition, subtraction, and multiplication:

Theorem 1.1 (Modular arithmetic rules)**.** *If* $a \equiv a' \pmod{N}$ *and* $b \equiv b' \pmod{N}$, *then:*

$$\begin{aligned} a + b &\equiv a' + b' \pmod{N} \\ a - b &\equiv a' - b' \pmod{N} \\ a \times b &\equiv a' \times b' \pmod{N} \end{aligned}$$

This means we can reduce intermediate results during computation, keeping numbers small.

Modular reduction in computation

Compute $3^{13} \bmod 7$ without computing $3^{13} = 1{,}594{,}323$ first.

$$\begin{aligned} 3^1 &\equiv 3 \pmod{7} \\ 3^2 &\equiv 9 \equiv 2 \pmod{7} \\ 3^4 &\equiv 2^2 = 4 \pmod{7} \\ 3^8 &\equiv 4^2 = 16 \equiv 2 \pmod{7} \\ 3^{13} &= 3^8 \cdot 3^4 \cdot 3^1 \equiv 2 \cdot 4 \cdot 3 = 24 \equiv 3 \pmod{7} \end{aligned}$$

We never had to deal with a number larger than 49!

Exam Tip

Modular arithmetic is important for understanding the GCD algorithm and primality testing. A typical exam question: **"Compute $a^k \bmod N$ using repeated squaring."** Always reduce at each step to keep numbers small.

Key Takeaway

Modular arithmetic keeps numbers in the range $\{0, 1, \ldots, N-1\}$, making large computations tractable.

1.7 Euclid's Algorithm for GCD

1.7.1 The Problem

The **greatest common divisor** (GCD) of two positive integers a and b is the largest integer that divides both a and b without remainder. For example, $\gcd(12, 8) = 4$ because $4 \mid 12$ and $4 \mid 8$, and no number larger than 4 divides both.

One approach: list all divisors of a and all divisors of b, and find the largest common one. This is correct but very slow for large numbers.

1.7.2 Euclid's Key Insight

Euclid noticed a beautiful property around 300 BCE:

Theorem 1.2 (Euclid's Rule)**.** *For positive integers $a \geq b > 0$:*

$$\gcd(a, b) = \gcd(b,\ a \bmod b).$$

Intuition

Suppose you want to tile a 1071×462 rectangle with identical square tiles, using the largest possible square. The answer is $\gcd(1071, 462)$.
Euclid's insight: instead of tiling the whole rectangle, you can first tile as many 462×462 squares as fit ($\lfloor 1071/462 \rfloor = 2$ of them), and then the problem reduces to tiling the leftover 462×147 strip. Repeat until the rectangle is a perfect square.

1.7.3 The Algorithm

Algorithm: Euclid(a, b)

```
Euclid(a, b):
    if b = 0: return a
    return Euclid(b, a mod b)
```

Trace of $\gcd(1071, 462)$

$$\begin{aligned}
\gcd(1071, 462) &= \gcd(462, 1071 \bmod 462) = \gcd(462, 147) \\
\gcd(462, 147) &= \gcd(147, 462 \bmod 147) = \gcd(147, 21) \\
\gcd(147, 21) &= \gcd(21, 147 \bmod 21) = \gcd(21, 0) \\
\gcd(21, 0) &= 21
\end{aligned}$$

Therefore $\gcd(1071, 462) = 21$. We needed only 3 steps.

Complexity Analysis

How many steps does Euclid's algorithm take?

Theorem 1.3. *Euclid's algorithm performs at most* $\mathcal{O}(\log(\min(a, b)))$ *steps.*

The key observation is that after two steps, the larger number at least halves. Specifically, $a \bmod b < a/2$ whenever $b \leq a/2$, and $a \bmod b = a - b < a/2$ whenever $b > a/2$. So after two steps, a is replaced by something less than $a/2$, giving $\mathcal{O}(\log a)$ total steps.

1.7.4 Extended Euclidean Algorithm

A powerful extension of Euclid's algorithm computes not just the GCD, but also two integers x and y such that:

$$a \cdot x + b \cdot y = \gcd(a, b).$$

This is called **Bezout's identity**.

Theorem 1.4 (Bezout's Identity)**.** *For any integers* a *and* b *(not both zero), there exist integers* x *and* y *such that* $ax + by = \gcd(a, b)$*.*

The extended algorithm works by unravelling the steps of Euclid's algorithm in reverse.

Extended GCD for $\gcd(35, 15)$

Running Euclid:

$$35 = 2 \times 15 + 5$$
$$15 = 3 \times 5 + 0$$

So $\gcd(35, 15) = 5$.
Back-substituting: $5 = 35 - 2 \times 15$. Thus $35 \cdot (1) + 15 \cdot (-2) = 5$, giving $x = 1$, $y = -2$.
Verify: $35(1) + 15(-2) = 35 - 30 = 5$. ✓

1.7.5 Modular Inverse

A number x is the **modular inverse** of a modulo N if $a \cdot x \equiv 1 \pmod{N}$. The extended GCD finds this: if $\gcd(a, N) = 1$, then the x from Bezout's identity satisfies $a \cdot x + N \cdot y = 1$, which means $a \cdot x \equiv 1 \pmod{N}$.

Modular inverse of 3 mod 7

$\gcd(3, 7) = 1$, and we can write $3 \cdot 5 + 7 \cdot (-2) = 15 - 14 = 1$. So $3 \cdot 5 \equiv 1 \pmod{7}$: the modular inverse of 3 mod 7 is 5.
Check: $3 \times 5 = 15 = 2 \times 7 + 1 \equiv 1 \pmod{7}$. ✓

1.7.6 Fast Modular Exponentiation

In cryptography, we frequently need to compute $a^k \bmod N$ for very large k. Computing this naively (multiply a by itself k times) takes $\mathcal{O}(k)$ multiplications, which is astronomical when k is, say, a 1024-bit number.

The key trick is **repeated squaring**: write k in binary and square repeatedly.

Algorithm: ModExp(x, y, N)

```
ModExp(x, y, N):
    if y = 0: return 1
    z = ModExp(x, ⌊y/2⌋, N)
    if y is even: return z^2 mod N
    else:         return x·z^2 mod N
```

Each recursive call halves k, so there are $\mathcal{O}(\log k)$ calls, each doing one multiplication. Total: $\mathcal{O}((\log k)(\log N)^2)$ bit operations (the $(\log N)^2$ accounts for the cost of multiplying two n-bit numbers).

1.7.7 Python Implementation

Python Code

```python
def gcd(a, b):
    while b != 0:
        a, b = b, a % b    # replace (a,b) with (b, a mod b)
    return a               # when b=0, the answer is a

def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0     # gcd=a, x=1, y=0: a*1 + 0*0 = a
    d, x1, y1 = extended_gcd(b, a % b)    # recurse
    x = y1
    y = x1 - (a // b) * y1                # back-substitute
    return d, x, y

def modular_inverse(a, N):
    d, x, _ = extended_gcd(a, N)
    if d != 1:
        raise ValueError("Inverse does not exist")
    return x % N          # ensure result is in {0,...,N-1}

def mod_exp(base, exp, mod):
    result = 1
    base = base % mod
    while exp > 0:
        if exp % 2 == 1:          # if current bit is 1
            result = result * base % mod
        exp //= 2                 # shift right (halve)
        base = base * base % mod # square the base
    return result
```

Try it: Trace `gcd(48, 18)` by hand. What are the successive pairs (a, b)?

Exam Tip

Euclid's algorithm is a guaranteed exam topic. Know how to:

1. Trace the algorithm step by step for a given (a, b).
2. State and apply Bezout's identity.
3. Find the modular inverse of a mod N using the extended GCD.
4. State the time complexity: $\mathcal{O}(\log(\min(a, b)))$ steps.

Key Takeaway

$\gcd(a, b) = \gcd(b, a \bmod b)$ — this single equation gives one of the oldest and most efficient algorithms in mathematics.

1.8 Primality Testing

A **prime number** is an integer $p \geq 2$ whose only divisors are 1 and p itself. Primes are the building blocks of the integers (Fundamental Theorem of Arithmetic), and they are central to modern cryptography. The key question: **given a large number N, is it prime?**

1.8.1 Trial Division

The simplest method: try to divide N by every integer from 2 up to $\sqrt{N}$. If any of them divide N evenly, then N is composite. Otherwise, N is prime.

Algorithm: IsPrime(N)

```
IsPrime(N):
    if N = 2: return true
    if N is even: return false
    for d = 3, 5, 7, ... while d² ≤ N:
        if d divides N: return false
    return true
```

Intuition

Why stop at $\sqrt{N}$? Because if $N = a \times b$ and both $a > \sqrt{N}$ and $b > \sqrt{N}$, then $a \times b > N$, a contradiction. So every composite N must have a factor $\leq \sqrt{N}$.

Is 97 prime?

We test divisors up to $\sqrt{97} \approx 9.8$, so we check $d = 3, 5, 7, 9$.

$$97 \div 3 = 32 \text{ remainder } 1 \quad \text{(not divisible)}$$
$$97 \div 5 = 19 \text{ remainder } 2 \quad \text{(not divisible)}$$
$$97 \div 7 = 13 \text{ remainder } 6 \quad \text{(not divisible)}$$
$$97 \div 9 = 10 \text{ remainder } 7 \quad \text{(not divisible)}$$

No divisor found $\Rightarrow$ 97 is prime.

Complexity: Trial division runs in $\mathcal{O}(\sqrt{N})$ steps. In terms of the bit-length $n = \log_2 N$ of the input, this is $\mathcal{O}(2^{n/2})$ — still exponential! For a 1024-bit number (used in cryptography), this is completely impractical.

1.8.2 Fermat's Little Theorem

A faster approach uses a beautiful theorem from number theory.

Theorem 1.5 (Fermat's Little Theorem)**.** *If p is prime and $1 \leq a \leq p-1$, then:*

$$a^{p-1} \equiv 1 \pmod{p}.$$

Intuition

This gives us a quick *test*: if N is prime, then for any a we should have $a^{N-1} \equiv 1 \pmod{N}$. If this fails for even one value of a, then N is definitely *not* prime. If it holds for many random values of a, then N is *probably* prime.

We call a a **Fermat witness** for the compositeness of N if $a^{N-1} \not\equiv 1 \pmod{N}$.

Fermat test for $N = 15$

Pick $a = 2$. Compute $2^{14} \bmod 15$:

$$2^{14} = 16384 = 1092 \times 15 + 4,$$

so $2^{14} \equiv 4 \pmod{15} \neq 1$.
Since $a^{N-1} \not\equiv 1 \pmod{N}$, we conclude: 15 is *definitely* composite. (Indeed, $15 = 3 \times 5$.)

1.8.3 The Fermat Primality Test

Algorithm: Primality(N, k)

```
Primality(N, k):
    for i = 1 to k:
        pick a at random from {1,...,N-1}
        if a^{N-1} mod N ≠ 1: return composite
    return prime (probably)
```

Each trial uses modular exponentiation, which takes $\mathcal{O}(\log N)$ multiplications. Total: $\mathcal{O}(k \log^2 N)$ bit operations. This is *polynomial* in $\log N$ — efficient even for huge numbers.

1.8.4 Carmichael Numbers: A Limitation

Warning

Fermat's test is not perfect. There exist **Carmichael numbers** — composite numbers N for which $a^{N-1} \equiv 1 \pmod{N}$ holds for *all* a with $\gcd(a, N) = 1$. The smallest Carmichael number is $561 = 3 \times 11 \times 17$. Fermat's test will never detect that 561 is composite (unless you happen to pick a sharing a factor with 561).

The *Miller-Rabin* test fixes this flaw (introduced in advanced cryptography courses), but for the purposes of this course, Fermat's test is sufficient to understand.

1.8.5 Python Implementation

Python Code

```python
import random

def is_prime_trial(n):
    if n < 2: return False
    if n == 2: return True
    if n % 2 == 0: return False      # even numbers > 2 are composite
    d = 3
    while d * d <= n:                # only need to check up to sqrt(n)
        if n % d == 0:
            return False             # found a divisor -- composite
        d += 2                       # skip even numbers
    return True                      # no divisor found -- prime

def mod_exp(base, exp, mod):         # fast modular exponentiation
    result = 1
    base = base % mod
    while exp > 0:
        if exp % 2 == 1:
            result = result * base % mod
        exp //= 2
        base = base * base % mod
    return result

def fermat_test(n, k=20):
    if n < 2: return False
    if n == 2 or n == 3: return True
    if n % 2 == 0: return False

    for _ in range(k):               # repeat k times
        a = random.randint(2, n - 2)
        if mod_exp(a, n - 1, n) != 1:
            return False             # Fermat witness found -- composite!
    return True                      # probably prime
```

Try it: Call `fermat_test(561, k=5)`. What happens? Why does 561 (a Carmichael number) fool the test?

Exam Tip

Expect questions of the form:

- **State Fermat's Little Theorem** and apply it to verify a specific case.
- **Why is trial division exponential?** (Because $\sqrt{N} = 2^{n/2}$ where $n = \log_2 N$.)
- **What is a Carmichael number?** (A composite number that passes Fermat's test for all valid bases.)
- **Compare trial division and Fermat's test** in terms of correctness and complexity.

Key Takeaway

Fermat's Little Theorem transforms primality testing from an exponential-time problem to a polynomial-time (probabilistic) one: check $a^{N-1} \equiv 1 \pmod{N}$ for several random a.

1.9 Chapter Summary

Concept / Algorithm	Time Complexity	Key Idea
Fibonacci (recursive)	$\mathcal{O}(\varphi^n)$	Exponential — impractical
Fibonacci (memoised)	$\mathcal{O}(n)$	Cache sub-results
Fibonacci (iterative)	$\mathcal{O}(n)$, space $\mathcal{O}(1)$	Keep last two values
Integer addition	$\mathcal{O}(n)$ bits	Bit-by-bit with carry
Integer multiplication	$\mathcal{O}(n^2)$ bits	n rows, each $2n$ bits
Euclid's GCD	$\mathcal{O}(\log \min(a, b))$	$\gcd(a, b) = \gcd(b, a \bmod b)$
Extended GCD	$\mathcal{O}(\log \min(a, b))$	Bezout: $ax + by = \gcd(a, b)$
Modular exponentiation	$\mathcal{O}((\log k) \log^2 N)$	Repeated squaring
Trial division (primality)	$\mathcal{O}(\sqrt{N})$	Check divisors up to $\sqrt{N}$
Fermat primality test	$\mathcal{O}(k \log^2 N)$	Fermat's Little Theorem

1.10 Practice Problems

Level 1 — Recall and Understand

1. Define an algorithm. What two properties does a good algorithm have?
2. Write down F_0 through F_{10}.
3. How many recursive calls does FIB(4) make? Draw the call tree.
4. What is the difference between memoisation and the iterative approach?
5. Simplify: $\mathcal{O}(5n^3 + 2n^2 + 100n + 7)$.
6. Simplify: $\mathcal{O}(n \log_3 n + n^{1.5})$.

7. Compute 1071 mod 462.
8. Compute 19 mod 7 and (-5) mod 7.

Level 2 — Apply

9. Trace $\gcd(252, 105)$ step by step using Euclid's algorithm. State each pair (a, b).
10. Use the extended GCD to find integers x, y such that $35x + 13y = \gcd(35, 13)$.
11. Find the modular inverse of 5 (mod 11).
12. Compute 3^{100} mod 17 using repeated squaring.
13. Apply Fermat's test with $a = 2$ to $N = 15$. Is 15 prime?
14. Use trial division to determine whether 97 is prime.

Level 3 — Prove

15. Prove that if $d \mid a$ and $d \mid b$, then $d \mid (a \bmod b)$. Use this to prove Euclid's Rule.
16. Prove that every composite number N has a prime factor $\leq \sqrt{N}$.
17. Prove that Euclid's algorithm terminates in at most $2\log_2(\min(a, b))$ steps.
18. If p is prime and $a^2 \equiv 1 \pmod{p}$, prove that $a \equiv \pm 1 \pmod{p}$.

Level 4 — Challenge

19. The Fibonacci numbers satisfy $\gcd(F_m, F_n) = F_{\gcd(m,n)}$. Verify this for $(m, n) = (8, 12)$.
20. Show that $561 = 3 \times 11 \times 17$ satisfies $a^{560} \equiv 1 \pmod{561}$ for all a with $\gcd(a, 561) = 1$ (hint: use Fermat's Little Theorem on each prime factor).

Chapter 2

Divide and Conquer, and Graph Search

Module II · 12 Hours · Sections 2.1, 2.2, 2.3, 3.1–3.3 of Dasgupta et al.

What you will learn in this chapter.

- The *divide and conquer* strategy and how it leads to fast algorithms.
- Karatsuba's algorithm: a clever way to multiply two large numbers faster.
- How to solve recurrence relations using the Master Theorem.
- Binary search and merge sort — two classic divide-and-conquer algorithms.
- What a graph is, and two ways to represent one in a computer.
- Depth-first search (DFS): how to systematically explore a graph.

2.1 The Divide and Conquer Strategy

Divide and conquer is a problem-solving technique that appears again and again in algorithm design. The idea is simple:

Intuition

Break a big problem into smaller pieces of the same type. Solve each piece (possibly by breaking it further). Combine the solutions to get the answer to the original problem.
Think of it like organising a large stack of papers. You split the stack in half, sort each half separately, and then merge the two sorted halves. Each half is easier to deal with than the whole.

Formally, a divide-and-conquer algorithm has three phases:

1. **Divide:** Split the input of size n into smaller subproblems, typically of size n/b for some $b \geq 2$.
2. **Conquer:** Solve each subproblem recursively.
3. **Combine:** Merge the solutions to form the overall answer.

The running time of such algorithms is described by a *recurrence relation*, which we will learn to solve in Section 2.3.

2.2 Fast Integer Multiplication: Karatsuba's Algorithm

Recall from Chapter 1 that the grade-school algorithm multiplies two n-digit numbers in $\mathcal{O}(n^2)$ steps: multiply the top number by each digit of the bottom, and add n such products.

Can we do better?

2.2.1 The Key Idea

Write two n-digit numbers x and y by splitting each in half:

$$x = x_L \cdot 10^{n/2} + x_R, \qquad y = y_L \cdot 10^{n/2} + y_R,$$

where x_L, x_R, y_L, y_R each have $n/2$ digits.

Then:

$$x \cdot y = x_L y_L \cdot 10^n + (x_L y_R + x_R y_L) \cdot 10^{n/2} + x_R y_R.$$

This involves four half-size multiplications: $x_L y_L$, $x_L y_R$, $x_R y_L$, $x_R y_R$. With four multiplications of size $n/2$, we get a recurrence $T(n) = 4T(n/2) + \mathcal{O}(n)$, which still gives $\mathcal{O}(n^2)$ (no improvement yet).

2.2.2 Karatsuba's Trick

Anatoly Karatsuba (1960) noticed that the middle term $x_L y_R + x_R y_L$ can be computed with just *one* multiplication instead of two, using:

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

We already compute $x_L y_L$ and $x_R y_R$, so the middle term costs only one *new* multiplication. Total multiplications: **3** (instead of 4).

The recurrence becomes $T(n) = 3T(n/2) + \mathcal{O}(n)$, which gives $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.585})$.

Karatsuba on $x = 1234$, $y = 5678$

Split: $x_L = 12$, $x_R = 34$, $y_L = 56$, $y_R = 78$.
Compute:

$$A = x_L \cdot y_L = 12 \times 56 = 672$$
$$B = x_R \cdot y_R = 34 \times 78 = 2652$$
$$C = (x_L + x_R)(y_L + y_R) = 46 \times 134 = 6164$$

Middle term: $C - A - B = 6164 - 672 - 2652 = 2840$.
Final:

$$x \cdot y = A \cdot 10^4 + (C - A - B) \cdot 10^2 + B = 6{,}720{,}000 + 284{,}000 + 2{,}652 = 7{,}006{,}652.$$

Verify: $1234 \times 5678 = 7{,}006{,}652$. ✓

Key Takeaway

Karatsuba's trick reduces n^2 to $n^{1.585}$ by computing three recursive products instead of four.

2.3 Solving Recurrences: The Master Theorem

Divide-and-conquer algorithms lead to **recurrence relations**: equations that define $T(n)$ in terms of $T(n/b)$ for smaller inputs. The **Master Theorem** gives us a direct way to read off the solution.

2.3.1 The Standard Form

Definition

A recurrence of the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \mathcal{O}(n^d)$$

where $a \geq 1$, $b > 1$, and $d \geq 0$ are constants, has the solution:

$$T(n) = \begin{cases} \mathcal{O}(n^d \log n) & \text{if } a = b^d \quad \textbf{(Case 1)} \\ \mathcal{O}(n^d) & \text{if } a < b^d \quad \textbf{(Case 2)} \\ \mathcal{O}(n^{\log_b a}) & \text{if } a > b^d \quad \textbf{(Case 3)} \end{cases}$$

2.3.2 Intuition for the Three Cases

At each level of recursion, the work done can be described as a geometric series. Let us think of a tree of subproblems. At the root (level 0): 1 problem of size n, doing n^d work. At level 1: a problems of size n/b, doing $a \cdot (n/b)^d = n^d \cdot (a/b^d)$ work. At level k: a^k problems of size n/b^k, doing $n^d \cdot (a/b^d)^k$ work.

Let $r = a/b^d$.

- If $r < 1$ (Case 2): work *decreases* each level. Most work is at the top: $\mathcal{O}(n^d)$.
- If $r = 1$ (Case 1): work is *the same* at every level. With $\log_b n$ levels: $\mathcal{O}(n^d \log n)$.
- If $r > 1$ (Case 3): work *increases* each level. Most work is at the leaves: $\mathcal{O}(n^{\log_b a})$.

Applying the Master Theorem

a) **Binary search:** $T(n) = T(n/2) + \mathcal{O}(1)$. Here $a = 1$, $b = 2$, $d = 0$. Check: $a = 1 = b^0 = 2^0$, so **Case 1**: $T(n) = \mathcal{O}(\log n)$.

b) **Merge sort:** $T(n) = 2T(n/2) + \mathcal{O}(n)$. Here $a = 2$, $b = 2$, $d = 1$. Check: $a = 2 = b^1 = 2^1$, so **Case 1**: $T(n) = \mathcal{O}(n \log n)$.

c) **Karatsuba:** $T(n) = 3T(n/2) + \mathcal{O}(n)$. Here $a = 3$, $b = 2$, $d = 1$. Check: $a = 3 > b^1 = 2$, so **Case 3**: $T(n) = \mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.585})$.

d) **Grade-school multiplication:** $T(n) = 4T(n/2) + \mathcal{O}(n)$. Here $a = 4$, $b = 2$, $d = 1$. Check: $a = 4 > b^1 = 2$, so **Case 3**: $T(n) = \mathcal{O}(n^{\log_2 4}) = \mathcal{O}(n^2)$.

Exam Tip

The Master Theorem is a very likely exam question. The steps to apply it:

1. Identify a, b, and d from the recurrence.
2. Compare a with b^d.
3. State the appropriate case and write the answer.

Be careful: the theorem applies to recurrences of the *exact* form $T(n) = a \cdot T(n/b) + \mathcal{O}(n^d)$.

Key Takeaway

The Master Theorem gives $T(n)$ in one step for recurrences of the form $aT(n/b) + \mathcal{O}(n^d)$. Compare a with b^d to choose the case.

2.4 Binary Search

Problem: Given a *sorted* array $A[1..n]$ and a value v, find the index i such that $A[i] = v$ (or determine that v is not in A).

Intuition

Think of looking up a word in a dictionary. You open to the middle. If the word on that page comes after your word, look in the left half; otherwise, look in the right half. Each step halves the remaining search space. After $\log_2 n$ steps, you either find the word or know it is absent.

Algorithm: BinarySearch(A, v, lo, hi)

```
BinarySearch(A, v, lo, hi):
    if lo > hi: return -1
    mid = ⌊(lo + hi)/2⌋
    if A[mid] = v: return mid
    if A[mid] > v: return BinarySearch(A, v, lo, mid - 1)
    else:          return BinarySearch(A, v, mid + 1, hi)
```

Binary search for $v = 7$ in $A = [2, 5, 7, 12, 18, 23, 31]$

$n = 7$.

- $\ell = 1, r = 7$: $m = 4$, $A[4] = 12 > 7$. Search left: $r = 3$.
- $\ell = 1, r = 3$: $m = 2$, $A[2] = 5 < 7$. Search right: $\ell = 3$.
- $\ell = 3, r = 3$: $m = 3$, $A[3] = 7 = v$. **Found at index 3.**

Only 3 comparisons for a list of 7 elements.

Complexity: Each step halves the array size. After k steps, the remaining size is $n/2^k$. When $n/2^k = 1$, we have $k = \log_2 n$. By the Master Theorem ($a = 1, b = 2, d = 0$, Case 1): $T(n) = \mathcal{O}(\log n)$.

Key Takeaway

Binary search finds an element in a sorted array in $\mathcal{O}(\log n)$ time by halving the search space at each step.

2.5 Merge Sort

Problem: Sort an array of n numbers in non-decreasing order.

2.5.1 The Algorithm

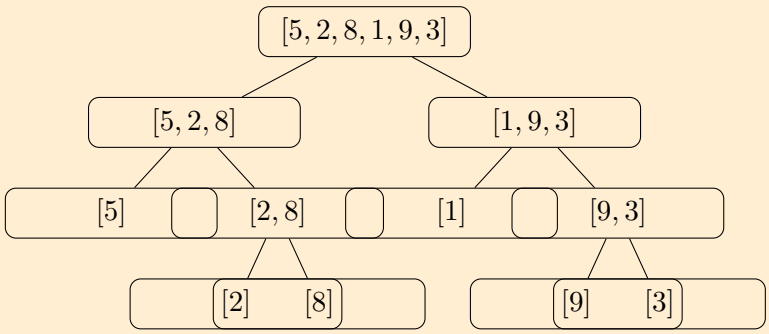
Algorithm: Mergesort($a[1..n]$)

```
Mergesort(a[1..n]):
    if n > 1:
        return Merge(Mergesort(a[1..⌊n/2⌋]),
                     Mergesort(a[⌊n/2⌋+1..n]))
    else:
        return a
```

The MERGE procedure takes two sorted arrays and combines them into one sorted array in $\mathcal{O}(n)$ time (compare the front elements of both arrays, take the smaller, repeat).

Merge sort on $[5, 2, 8, 1, 9, 3]$

Step 1: Divide until we reach arrays of size 1:



Step 2: Merge back up:

$$\begin{aligned}
\text{Merge}([2], [8]) &= [2, 8] \\
\text{Merge}([5], [2, 8]) &= [2, 5, 8] \\
\text{Merge}([3], [9]) &= [3, 9] \\
\text{Merge}([1], [3, 9]) &= [1, 3, 9] \\
\text{Merge}([2, 5, 8], [1, 3, 9]) &= [1, 2, 3, 5, 8, 9]
\end{aligned}$$

2.5.2 Complexity Analysis

The recurrence is $T(n) = 2T(n/2) + \mathcal{O}(n)$ (split into 2 halves; merge takes linear time). By Master Theorem ($a = 2, b = 2, d = 1$: $a = b^d$ so Case 1): $T(n) = \mathcal{O}(n \log n)$.

Algorithm	Best	Average	Worst	Space
Merge sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Quick sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$
Insertion sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$

Merge sort's $\mathcal{O}(n \log n)$ in all cases makes it a reliable choice, though it requires $\mathcal{O}(n)$ extra memory for the merged arrays.

Exam Tip

You may be asked to:

- Trace merge sort on a given array.
- Derive the recurrence $T(n) = 2T(n/2)+\mathcal{O}(n)$ and apply the Master Theorem.
- Compare merge sort with other sorting algorithms.

Key Takeaway

Merge sort achieves $\mathcal{O}(n \log n)$ in all cases by dividing the array in half, recursively sorting each half, and merging in linear time.

2.6 Introduction to Graphs

2.6.1 What Is a Graph?

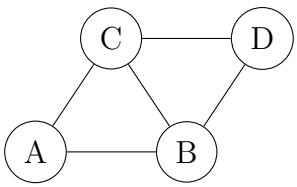
Many problems in computing involve *relationships* between objects: cities and roads, webpages and links, people and friendships, courses and prerequisites. The mathematical structure that captures all of these is a **graph**.

Definition

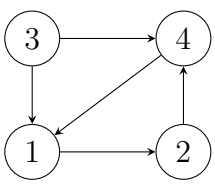
A **graph** $G = (V, E)$ consists of:

- A set V of **vertices** (also called *nodes*).
- A set E of **edges** (also called *links*) — pairs of vertices.

An **undirected** graph has edges $\{u, v\}$ that go both ways. A **directed** graph (or **digraph**) has edges (u, v) with a direction: from u to v.



Undirected graph



Directed graph

2.6.2 Basic Terminology

Definition

- The **degree** of a vertex v in an undirected graph is the number of edges touching v.
- In a directed graph, the **in-degree** of v is the number of edges *into* v, and the **out-degree** is the number of edges *out of* v.
- A **path** from u to v is a sequence of vertices $u = v_0, v_1, \ldots, v_k = v$ where each consecutive pair is connected by an edge.
- A **cycle** is a path that starts and ends at the same vertex (with at least one edge).
- A graph is **connected** if there is a path between every pair of vertices.
- A **tree** is a connected graph with no cycles. A tree with n vertices has exactly $n - 1$ edges.
- $|V|$ denotes the number of vertices; $|E|$ denotes the number of edges.

A simple graph and its properties

Graph G: vertices $\{A, B, C, D, E\}$, edges $\{AB, AC, BC, BD, CD, DE\}$.

- Degrees: $\deg(A) = 2$, $\deg(B) = 3$, $\deg(C) = 3$, $\deg(D) = 3$, $\deg(E) = 1$.
- Is there a path from A to E? Yes: $A \to B \to D \to E$.
- Is the graph connected? Yes (you can reach any vertex from any other).
- Does it have a cycle? Yes: $A \to B \to C \to A$.

2.7 Representing Graphs in a Computer

To work with a graph algorithmically, we need to store it in a computer. Two standard representations:

2.7.1 Adjacency Matrix

A $|V| \times |V|$ matrix M where $M[u][v] = 1$ if there is an edge from u to v, and 0 otherwise.

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	0
D	0	1	1	0	1
E	0	0	0	1	0

Space: $\mathcal{O}(|V|^2)$ — even if most entries are 0.
Checking if (u, v) is an edge: $\mathcal{O}(1)$ — just look at $M[u][v]$.
Listing all neighbours of v: $\mathcal{O}(|V|)$ — scan the whole row.

2.7.2 Adjacency List

For each vertex v, store a list of its neighbours.

$$\begin{aligned} A&: [B, C] \\ B&: [A, C, D] \\ C&: [A, B, D] \\ D&: [B, C, E] \\ E&: [D] \end{aligned}$$

Space: $\mathcal{O}(|V| + |E|)$ — proportional to what is actually there.
Listing all neighbours of v: $\mathcal{O}(\deg(v))$ — just read the list.
Checking if (u, v) is an edge: $\mathcal{O}(\deg(u))$ — scan u's neighbour list.

2.7.3 Which to Use?

Property	Adjacency Matrix	Adjacency List
Space	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(\|V\| + \|E\|)$
Edge lookup	$\mathcal{O}(1)$	$\mathcal{O}(\deg(v))$
Neighbour iteration	$\mathcal{O}(\|V\|)$	$\mathcal{O}(\deg(v))$
Best for	Dense graphs ($\|E\| \approx \|V\|^2$)	Sparse graphs

Intuition

In real applications, most graphs are sparse: cities have roads to nearby cities, not all cities. In sparse graphs, $|E| \ll |V|^2$, so adjacency lists are much more space-efficient. **We will use adjacency lists throughout this course.**

In Python, an adjacency list is represented as a dictionary:

Python Code

```python
# Undirected graph: each vertex maps to a list of its neighbours
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D'],
    'D': ['B', 'C', 'E'],
    'E': ['D']
}
# For a directed graph, just omit the reverse edges
```

Exam Tip

Be able to:

- Draw the adjacency matrix for a given graph (and vice versa).
- Draw the adjacency list for a given graph (and vice versa).
- State the space and time complexity of each representation.

2.8 Depth-First Search (DFS)

2.8.1 The Idea

Depth-First Search is a systematic way to visit all vertices in a graph. Think of exploring a maze: you always go as deep as possible along one path, and only backtrack when you reach a dead end.

Intuition

Start at a vertex. Go to an unvisited neighbour. Then go to one of *its* unvisited neighbours. Keep going deeper. When you're stuck (no unvisited neighbours), backtrack one step and try another direction. This is exactly how you would explore a cave system: always push deeper before turning back.

2.8.2 Pre- and Post-visit Times

As we run DFS, we record two timestamps for each vertex v:

- pre[v]: the time we *first arrive* at v.
- post[v]: the time we *finish exploring* v (all neighbours done, backtracking).

A global clock starts at 1 and ticks up by 1 each time we record a time.

Algorithm: DFS(G)

```
DFS(G):
    for all v ∈ V: visited(v) = false
    clock = 1
    for all v ∈ V:
        if not visited(v): Explore(v)

Explore(v):
    visited(v) = true
    pre[v] = clock; clock = clock + 1
    for each edge (v, u) ∈ E:
        if not visited(u): Explore(u)
    post[v] = clock; clock = clock + 1
```

DFS trace on G

Suppose we start at A and process neighbours alphabetically.

Vertex	pre	post	Parent	Action
A	1	10	—	Start exploring
B	2	7	A	From A, go to B
C	3	6	B	From B, A visited; go to C
D	4	5	C	From C, A, B visited; go to D
D	4	5	C	E from D
E	5	6	D	From D, go to E

(Actual trace depends on adjacency list order. The key point: each vertex gets exactly one pre and one post time.)

2.8.3 Edge Classification in Directed Graphs

When running DFS on a directed graph, we can classify each edge (u, v):

Edge type	Condition	Meaning
Tree edge	v not yet visited when (u, v) explored	v discovered through u
Back edge	$\text{pre}[v] < \text{pre}[u] < \text{post}[u] < \text{post}[v]$	Creates a cycle
Forward edge	v descendant of u in DFS tree	Shortcut forward
Cross edge	Everything else	No ancestor relation

Theorem 2.1. *A directed graph has a cycle if and only if DFS reveals a* ***back edge****.*

Intuition

A back edge goes from a vertex u to an ancestor v — meaning we can start at v, reach u, and then return to v, forming a loop.

2.8.4 Properties of Pre/Post Times

The pre/post intervals have a beautiful *nesting* property:

Theorem 2.2 (Nesting Property)**.** *For any two vertices u and v in a DFS:*

- *The intervals $[pre[u], post[u]]$ and $[pre[v], post[v]]$ are either* nested *(one inside the other) or* disjoint*.*
- *They are nested iff one is an ancestor of the other in the DFS tree.*

2.8.5 DFS on Disconnected Graphs

If the graph is not connected, a single call to Explore will only visit the component reachable from the starting vertex. The outer loop in DFS handles this by starting a new exploration from any unvisited vertex. Each time the outer loop starts a new Explore, it discovers a new **connected component**.

2.8.6 Complexity

DFS visits each vertex exactly once (sets it as visited when first encountered) and examines each edge at most twice (once from each end in an undirected graph, or once in a directed graph). Total:

$$\text{DFS runs in } \mathcal{O}(|V| + |E|).$$

2.8.7 Python Implementation

Python Code

```python
import sys
sys.setrecursionlimit(10000)  # Python's default limit is 1000

def dfs(graph):
    visited = set()
    pre = {}          # pre[v] = time when v is first entered
    post = {}         # post[v] = time when v is finished
    tree = {}         # tree[v] = parent of v in DFS tree (None if
        root)
    clock = [1]       # use a list so inner function can modify it
    components = []   # list of connected components found

    def explore(v, parent):
        visited.add(v)
        pre[v] = clock[0]; clock[0] += 1  # stamp entry time
        tree[v] = parent
        for u in graph.get(v, []):
            if u not in visited:
                explore(u, v)
        post[v] = clock[0]; clock[0] += 1 # stamp exit time

    for v in graph:
        if v not in visited:
            comp_before = set(visited)
            explore(v, None)
            comp = set(visited) - comp_before
            comp.add(v)
            components.append(frozenset(comp))

    return pre, post, tree, components

def has_cycle_directed(graph):
    visited = set()
    on_stack = set()    # vertices in current DFS call stack
    def dfs_visit(v):
        visited.add(v); on_stack.add(v)
        for u in graph.get(v, []):
            if u not in visited:
                if dfs_visit(u): return True
            elif u in on_stack:
                return True     # back edge found!
        on_stack.remove(v)
        return False
    for v in graph:
        if v not in visited:
            if dfs_visit(v): return True
    return False

def topological_sort(graph):
    if has_cycle_directed(graph):
        raise ValueError("Graph has a cycle")
    visited = set(); order = []
    def explore(v):
```

```
        visited.add(v)
        for u in graph.get(v, []):
            if u not in visited: explore(u)
        order.append(v)    # add to order AFTER all successors visited
    for v in graph:
        if v not in visited: explore(v)
    return list(reversed(order))  # reverse post-order = topological order
```

Try it: Run `topological_sort` on the prerequisite DAG: `Calculus -> ODE -> Numerical` and `Calculus -> LinAlg -> Numerical`.

Exam Tip

DFS is one of the most examined topics. Typical questions:

- **Trace DFS** on a given graph, listing pre and post times for each vertex.
- **Classify each edge** as tree, back, forward, or cross.
- **State the complexity** of DFS: $\mathcal{O}(|V| + |E|)$.
- **How does DFS detect cycles?** (Back edges in directed; back edges in undirected.)

Key Takeaway

DFS explores a graph in $\mathcal{O}(|V| + |E|)$ time, assigning pre/post timestamps that reveal the tree structure, cycles, and connected components.

2.9 Chapter Summary

Algorithm / Concept	Complexity	Key Idea
Karatsuba multiplication	$\mathcal{O}(n^{1.585})$	3 sub-multiplications instead of 4
Binary search	$\mathcal{O}(\log n)$	Halve search space each step
Merge sort	$\mathcal{O}(n \log n)$	Divide, sort halves, merge
Adjacency matrix	Space $\mathcal{O}(\|V\|^2)$	Edge lookup $\mathcal{O}(1)$
Adjacency list	Space $\mathcal{O}(\|V\| + \|E\|)$	Efficient for sparse graphs
DFS	$\mathcal{O}(\|V\| + \|E\|)$	Explore deeply, backtrack
Back edge detection	within DFS	Indicates a cycle

Master Theorem reference: $T(n) = aT(n/b) + \mathcal{O}(n^d)$:

- $a = b^d$: $T(n) = \mathcal{O}(n^d \log n)$
- $a < b^d$: $T(n) = \mathcal{O}(n^d)$
- $a > b^d$: $T(n) = \mathcal{O}(n^{\log_b a})$

2.10 Practice Problems

Level 1 — Recall and Understand

1. State the three phases of divide and conquer.
2. What is a recurrence relation? Give one example.
3. What is the key insight in Karatsuba's algorithm that reduces 4 multiplications to 3?
4. What are the three cases of the Master Theorem? State the condition for each case.
5. Define: vertex, edge, degree, path, cycle, connected graph, tree.
6. What is the difference between an adjacency matrix and an adjacency list?
7. What do pre and post times in DFS represent?
8. What type of edge in DFS indicates a cycle in a directed graph?

Level 2 — Apply

9. Use the Master Theorem to solve: (a) $T(n) = 4T(n/2) + \mathcal{O}(n)$; (b) $T(n) = 2T(n/3) + \mathcal{O}(1)$; (c) $T(n) = 3T(n/3) + \mathcal{O}(n)$.
10. Trace binary search for the value 13 in $[2, 5, 7, 11, 13, 17, 19, 23]$. List each comparison.
11. Trace merge sort on $[3, 1, 4, 1, 5, 9, 2, 6]$. Show all levels of the recursion tree.
12. Draw the adjacency matrix and adjacency list for: $V = \{1, 2, 3, 4\}$, $E = \{(1, 2), (2, 3), (3, 4), (4, 1), ($
13. Run DFS on the graph in Problem 12, starting from vertex 1. Record pre and post times for each vertex.
14. Identify all edge types (tree/back/forward/cross) in the DFS of Problem 13, treating the graph as directed.

Level 3 — Prove

15. Prove that a tree with n vertices has exactly $n - 1$ edges.
16. Prove the nesting property: in a DFS, the pre/post intervals of any two vertices are either nested or disjoint.
17. Prove that the Merge step of merge sort takes $\mathcal{O}(n)$ time for two sorted arrays of total size n.

Level 4 — Challenge

18. The Fibonacci recurrence is $F(n) = F(n - 1) + F(n - 2)$. This does *not* fit the Master Theorem (not of the form $aT(n/b)$). However, one can show $F(n) = \mathcal{O}(\varphi^n)$. Explain why the Master Theorem does not apply.

Chapter 3

Graph Algorithms

Module III · 12 Hours · Sections 3.4, 4.1–4.5, 4.7 of Dasgupta et al.

What you will learn in this chapter.

- How to check whether a graph is connected, and find all connected components.
- What directed acyclic graphs (DAGs) are and why they matter.
- Topological ordering: scheduling tasks that have prerequisites.
- Strongly connected components and Kosaraju's two-pass algorithm.
- Breadth-first search (BFS) and shortest paths in unweighted graphs.
- Weighted graphs and Dijkstra's algorithm for shortest paths.
- Priority queues and their implementations.
- Shortest paths in DAGs using topological order.

3.1 Connectivity

A graph is **connected** if you can reach any vertex from any other vertex by following edges. But many real-world graphs are not connected: perhaps a country has some isolated islands not connected by roads.

Definition

A **connected component** of an undirected graph is a maximal set of vertices such that every pair of vertices in the set is connected by a path. "Maximal" means we cannot add any more vertices without breaking the connectivity.

Intuition

Think of a city's road network during a flood. Some roads are submerged, splitting the city into isolated neighbourhoods. Each neighbourhood is a connected component — you can drive anywhere within it, but cannot reach another neighbourhood.

3.1.1 Algorithm: Finding All Components

We use DFS. Each time the outer loop of DFS starts a new Explore from an unvisited vertex, it discovers a new component.

Algorithm: ConnectedComponents(G)

```
ConnectedComponents(G):
    for all v ∈ V: visited(v) = false
    label = 0
    for all v ∈ V:
        if not visited(v):
            label = label + 1
            Explore(v, label)

Explore(v, label):
    comp[v] = label; visited(v) = true
    for each neighbour u of v:
        if not visited(u): Explore(u, label)
```

Complexity: $\mathcal{O}(|V| + |E|)$ — DFS touches each vertex and edge once.

Finding components

Graph: $V = \{A, B, C, D, E, F\}$, edges: $\{AB, BC, DE\}$.
DFS from A: visits A, B, C. Component 1.
D not visited: DFS from D: visits D, E. Component 2.
F not visited: DFS from F: visits F. Component 3.
Result: three components: $\{A, B, C\}$, $\{D, E\}$, $\{F\}$.

Exam Tip

A graph is connected iff the component-finding algorithm finds exactly **one** component. You may be asked to verify connectivity or count components on a given graph.

3.2 Directed Acyclic Graphs (DAGs)

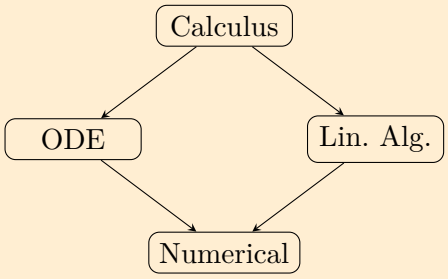
Definition

A **directed acyclic graph (DAG)** is a directed graph with no directed cycles. That is, starting from any vertex and following edges in their direction, you can never return to the vertex you started from.

Intuition

Course prerequisites are a perfect example. "Calculus must be taken before Differential Equations." There is no way for a course to (directly or indirectly) be its own prerequisite — that would be a cycle, which is impossible in practice. DAGs model any situation with tasks that have dependencies but no circular dependencies.

A course prerequisite DAG



No cycles: you cannot take Numerical Analysis before its prerequisites.

Detecting DAGs: A directed graph is a DAG if and only if DFS finds no back edges. (Recall from Chapter 2: back edges create cycles.)

3.3 Topological Ordering

Definition

A **topological ordering** (or topological sort) of a DAG is a linear ordering of all vertices such that for every directed edge (u, v), u appears before v in the ordering.

Intuition

Think of a to-do list where some tasks depend on others. A topological ordering is an order in which you can complete all tasks without ever working on a task whose prerequisite is not yet done.

Theorem 3.1. *Every DAG has at least one topological ordering. Moreover, a directed graph has a topological ordering if and only if it is a DAG.*

3.3.1 Algorithm: DFS-based Topological Sort

The topological order is simply the *reverse* of the DFS finishing (post) order.

Algorithm: TopologicalSort(G)

```
TopologicalSort(G):
    Run DFS on G, recording post times
    Output vertices in decreasing order of post time
```

Why does this work? For any edge (u, v) in a DAG (not a back edge), post$[u]$ > post$[v]$. (We finish exploring from u after we finish exploring v, because v is a successor of u.) So ordering by decreasing post time gives a valid topological order.

Topological sort of the course prerequisite DAG

Suppose DFS gives post times:

Vertex	pre	post
Calculus	1	8
ODE	2	5
Numerical	3	4
LinAlg	6	7

Sorted by decreasing post: **Calculus** → **LinAlg** → **ODE** → **Numerical**. This is a valid order: each course appears after all its prerequisites.

Exam Tip

The exam may ask you to find a topological order for a given DAG. Quick method (no DFS needed for simple examples): repeatedly remove a vertex with no incoming edges. The order of removal is a topological order.

Key Takeaway

The topological order of a DAG is the reverse DFS finishing order. It exists iff the graph is acyclic.

3.4 Strongly Connected Components

3.4.1 What Are SCCs?

In an undirected graph, a connected component is a maximal set of mutually reachable vertices. In a directed graph, we need a stronger notion: we want vertices that can reach *each other* (in both directions).

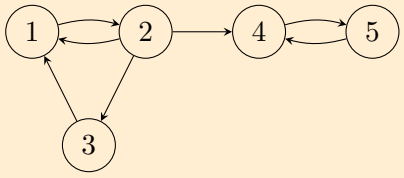
Definition

A **strongly connected component (SCC)** of a directed graph is a maximal set S of vertices such that for every pair $u, v \in S$, there is a directed path from u to v *and* a directed path from v to u.

Intuition

Think of a city's one-way street network. Two locations are in the same SCC if you can drive from one to the other (and back) using only one-way streets.

SCCs in a directed graph



SCCs: $\{1, 2, 3\}$ (all can reach each other via $1 \to 2 \to 3 \to 1$), and $\{4, 5\}$ (form a 2-cycle).

3.4.2 Kosaraju's Algorithm

Kosaraju's algorithm finds all SCCs in $\mathcal{O}(|V| + |E|)$ using two DFS passes.

Algorithm: SCC(G)

```
SCC(G):
    Run DFS on G; record vertices in decreasing post-time
        order
    Construct G^R (reverse all edge directions)
    Run DFS on G^R processing vertices in that order
    Each DFS tree in the second pass is one SCC
```

Intuition

Why does this work? The vertex with the highest post time in DFS is a "source" SCC (no edges coming in from other SCCs). On the reversed graph, this vertex can only reach its own SCC. So the DFS tree rooted at it (in Pass 2 on G^R) gives exactly one SCC. We then repeat for the next highest-post vertex, and so on.

Kosaraju on a 5-vertex graph

G: edges 1→2, 2→3, 3→1, 2→4, 4→5, 5→4.
Pass 1 (DFS on G): suppose post times are 3:6, 1:7, 2:8, 4:3, 5:4. Decreasing post order: $2, 1, 5, 4, 3$.
Reverse graph G^R: edges 2→1, 3→2, 1→3, 4→2, 5→4, 4→5.
Pass 2 (DFS on G^R in order $2, 1, 5, 4, 3$):

- Start from 2: reaches $\{1, 2, 3\}$. **SCC 1** $= \{1, 2, 3\}$.
- Start from 5: reaches $\{4, 5\}$. **SCC 2** $= \{4, 5\}$.

Complexity: Two DFS passes + one graph reversal $= \mathcal{O}(|V| + |E|)$.

Exam Tip

Expect a step-by-step trace of Kosaraju's algorithm on a small graph (5–7 vertices). Be sure you can:

1. Run DFS, recording post times.

2. Reverse the graph.
3. Run DFS on the reversed graph in decreasing post order.
4. Identify each DFS tree as one SCC.

Key Takeaway

Kosaraju's algorithm finds all SCCs in $\mathcal{O}(|V| + |E|)$ using two DFS passes: one on the original graph, one on the reversed graph.

3.5 Breadth-First Search (BFS)

3.5.1 The Idea

Where DFS goes deep before wide, BFS explores the graph *level by level*. Starting from a source vertex s, BFS first visits all vertices at distance 1, then all at distance 2, and so on.

Intuition

Imagine dropping a stone in a pond. The ripples spread outward in concentric circles. BFS is like those ripples: first all vertices one hop away, then two hops, then three.

Algorithm: BFS(G, s)

```
BFS(G, s):
    for all u ∈ V: dist(u) = ∞
    dist(s) = 0
    Q = [s]     [queue containing just s]
    while Q is not empty:
        u = eject(Q)
        for all edges (u, v) ∈ E:
            if dist(v) = ∞:
                inject(v, Q)
                dist(v) = dist(u) + 1
```

BFS from vertex A

Graph: A–B, A–C, B–D, C–D, D–E.

Step	Dequeue	Queue after	Distances updated
1	A (dist=0)	$[B, C]$	dist$[B]$=1, dist$[C]$=1
2	B (dist=1)	$[C, D]$	dist$[D]$=2 (from B)
3	C (dist=1)	$[D]$	D already visited
4	D (dist=2)	$[E]$	dist$[E]$=3
5	E (dist=3)	[]	Done

Final distances from A: $B : 1$, $C : 1$, $D : 2$, $E : 3$.

3.5.2 Shortest Paths (Unweighted Graphs)

Theorem 3.2. *BFS computes the shortest path (in terms of number of edges) from the source s to every reachable vertex.*

To reconstruct the actual path (not just the distance), we record a *parent* pointer when we first discover each vertex. Then to find the path from s to t, we follow parent pointers backwards from t to s.

Complexity: Each vertex is enqueued once and dequeued once. Each edge is examined at most twice (once from each endpoint in an undirected graph). Total: $\mathcal{O}(|V| + |E|)$.

Warning

BFS finds shortest paths only in *unweighted* graphs (or equivalently, graphs where every edge has weight 1). For weighted graphs, use Dijkstra's algorithm (Section 3.7).

3.5.3 Python Implementation

Python Code

```python
from collections import deque

def bfs(graph, source):
    dist = {v: float('inf') for v in graph}  # all distances start as infinity
    parent = {v: None for v in graph}
    dist[source] = 0
    queue = deque([source])                   # use a deque for O(1) popleft

    while queue:
        v = queue.popleft()
        for u in graph.get(v, []):
            if dist[u] == float('inf'):       # u not yet visited
                dist[u] = dist[v] + 1
                parent[u] = v
                queue.append(u)

    return dist, parent

def shortest_path(parent, source, target):
    if parent.get(target) is None and target != source:
        return None      # target unreachable
    path = []
    v = target
    while v is not None:
        path.append(v)
        v = parent.get(v)
    path.reverse()       # we built it backwards
    return path
```

Try it: Build the graph from the worked example above and call `bfs(graph, 'A')`. Verify the distances.

Exam Tip

BFS questions:

- Trace BFS from a given source, listing the queue state after each step.
- Find the shortest path between two vertices using BFS.
- Compare BFS and DFS: BFS finds shortest paths; DFS reveals depth structure and cycles.

Key Takeaway

BFS explores a graph level by level using a queue, computing shortest hop-distances in $\mathcal{O}(|V| + |E|)$.

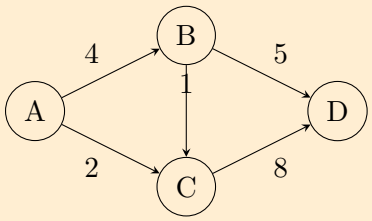
3.6 Weighted Graphs and the Shortest Path Problem

In many real problems, edges have associated **weights** (costs, distances, travel times).

Definition

A **weighted graph** is a graph $G = (V, E, w)$ where $w : E \to \mathbb{R}$ assigns a real-valued **weight** to each edge.
The **weight** (or length) of a path $v_0, v_1, \ldots, v_k$ is $\sum_{i=0}^{k-1} w(v_i, v_{i+1})$.
The **shortest path** from s to t is the path of minimum total weight.

A weighted graph



Paths from A to D: $A \to B \to D$: weight $4 + 5 = 9$. $A \to C \to D$: weight $2 + 8 = 10$. $A \to B \to C \to D$: weight $4 + 1 + 8 = 13$. Shortest: $A \to B \to D$ with weight 9.

Note: BFS would give the *hop-count* shortest path ($A \to B \to D$ or $A \to C \to D$, both 2 hops), but cannot account for the different edge weights. We need a smarter algorithm.

3.7 Dijkstra's Algorithm

3.7.1 The Main Idea

Dijkstra's algorithm (Edsger Dijkstra, 1956) finds shortest paths in weighted graphs with *non-negative* edge weights.

Intuition

Imagine you are navigating a city by car, starting from your hotel. You maintain a list of "known shortest distances" to each destination. At each step, you select the unvisited destination that you know the shortest distance to, visit it, and update the distances to its neighbours (if going through this city gives a shorter route). This greedy strategy works because once you have found the shortest distance to a city, no future discovery can make it shorter (all weights are non-negative).

3.7.2 The Algorithm

Algorithm: Dijkstra(G, l, s)

```
Dijkstra(G, l, s):
    for all u ∈ V:
        dist(u) = ∞
        prev(u) = nil
    dist(s) = 0
    H = makequeue(V)     [priority queue keyed by dist
       values]
    while H is not empty:
        u = deletemin(H)
        for all edges (u, v) ∈ E:
            if dist(u) + l(u, v) < dist(v):
                dist(v) = dist(u) + l(u, v)
                prev(v) = u
                decreasekey(H, v)
```

The operation $\text{dist}[v] \leftarrow \text{dist}[u] + w$ is called **edge relaxation**: we check whether going through u gives a better (shorter) route to v.

3.7.3 Step-by-Step Trace

Dijkstra from A on the graph above

Graph: $A \to B$ (wt 4), $A \to C$ (wt 2), $B \to C$ (wt 1), $B \to D$ (wt 5), $C \to D$ (wt 8).

Step	Extract	dist[A]	dist[B]	dist[C]	dist[D]
Init	—	0	∞	∞	∞
1	A (0)	0	**4**	**2**	∞
2	C (2)	0	**3**	2	**10**
3	B (3)	0	3	2	**8**
4	D (8)	0	3	2	8

Boldface = newly updated values.
Shortest distances: $A : 0$, $B : 3$, $C : 2$, $D : 8$.
Shortest paths: A; $A \to C \to B$; $A \to C$; $A \to C \to B \to D$.

3.7.4 Correctness

Theorem 3.3. *When Dijkstra's algorithm extracts vertex u from the priority queue, dist[u] is the true shortest distance from s to u.*

Intuition

Proof sketch by induction. The source s is extracted first with distance 0 (correct, by definition). Suppose all previously extracted vertices have correct distances. The next extracted vertex u has the current minimum $\text{dist}[u]$. Could there be a shorter path through some unvisited vertex x? No: any such path must pass through some unvisited vertex x with $\text{dist}[x] \geq \text{dist}[u]$ (since u has the minimum), and since all weights are non-negative, the extension beyond x cannot reduce the distance below $\text{dist}[u]$.

Warning

Dijkstra's algorithm *fails* (gives wrong answers) when edge weights are negative. For graphs with negative weights, use the Bellman-Ford algorithm (not in this syllabus). For DAGs with negative weights, use the DAG shortest path algorithm (Section 3.9).

3.7.5 Python Implementation

Python Code

```python
import heapq

def dijkstra(graph, source):
    # graph[u] = list of (neighbour, weight) pairs
    dist = {v: float('inf') for v in graph}
    prev = {v: None for v in graph}   # for path reconstruction
    dist[source] = 0

    pq = [(0, source)]                # (distance, vertex) -- min-heap
    visited = set()

    while pq:
        d, u = heapq.heappop(pq)      # extract minimum-distance vertex
        if u in visited:
            continue                  # skip stale entries in heap
        visited.add(u)

        for v, weight in graph.get(u, []):
            if v not in visited:
                new_dist = dist[u] + weight
                if new_dist < dist[v]:    # edge relaxation
                    dist[v] = new_dist
                    prev[v] = u
                    heapq.heappush(pq, (new_dist, v))

    return dist, prev

def get_path(prev, source, target):
    if prev.get(target) is None and target != source:
```

```
        return None    # unreachable
    path = []
    v = target
    while v is not None:
        path.append(v)
        v = prev.get(v)
    path.reverse()
    return path
```

Try it: Build the graph from the worked example and verify `dijkstra(graph, 'A')` returns the distances computed in the trace.

Exam Tip

Dijkstra is a key exam topic:

1. Trace the algorithm step by step on a given graph, showing the distance table after each extraction.
2. Identify the shortest path from source to a target by following the *prev* pointers.
3. State why Dijkstra fails for negative weights.
4. State the complexity: $\mathcal{O}((|V| + |E|) \log |V|)$ with a binary heap.

Key Takeaway

Dijkstra's algorithm greedily extracts the closest unvisited vertex and relaxes its outgoing edges. It requires non-negative weights and runs in $\mathcal{O}((|V| + |E|) \log |V|)$ with a binary heap.

3.8 Priority Queue Implementations

Dijkstra's efficiency depends on how we implement the **priority queue** — a data structure that supports:

1. **Insert**(v, key): add vertex v with priority key.
2. **ExtractMin**: remove and return the vertex with the smallest key.
3. **DecreaseKey**$(v, \text{new\_key})$: reduce the key of vertex v.

Implementation	Insert	ExtractMin	DecreaseKey
Unsorted array	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|)$	$\mathcal{O}(1)$
Sorted array	$\mathcal{O}(\|V\|)$	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|)$
Binary heap	$\mathcal{O}(\log \|V\|)$	$\mathcal{O}(\log \|V\|)$	$\mathcal{O}(\log \|V\|)$
Fibonacci heap	$\mathcal{O}(1)$ amortised	$\mathcal{O}(\log \|V\|)$ amortised	$\mathcal{O}(1)$ amortised

Priority Queue	Dijkstra Total Time	Best for
Unsorted array	$\mathcal{O}(\|V\|^2)$	Dense graphs ($\|E\| \approx \|V\|^2$)
Binary heap	$\mathcal{O}((\|V\| + \|E\|) \log \|V\|)$	Sparse graphs
Fibonacci heap	$\mathcal{O}(\|V\| \log \|V\| + \|E\|)$	Theoretical optimum

Intuition

For most practical purposes, a binary heap is the right choice. The Fibonacci heap is theoretically optimal but complex to implement. Python's `heapq` module provides a binary min-heap.

Exam Tip

Know the three implementations (unsorted array, binary heap, Fibonacci heap), their operation complexities, and the resulting total complexity for Dijkstra.

3.9 Shortest Paths in DAGs

When the graph is a DAG, we can find shortest paths even faster than Dijkstra — and we can handle *negative* edge weights!

Intuition

In a DAG, there is a topological order: we can process vertices from left to right. When we process vertex u, all vertices that could *feed into* u have already been processed (since they come before u in topological order). So we can relax all outgoing edges of u immediately, knowing the distance to u is already correct.

Algorithm: DAGshortestpath(G, l, s)

```
DAGshortestpath(G, l, s):
    Linearise G (topological sort)
    for all u ∈ V: dist(u) = ∞
    dist(s) = 0
    for each u ∈ V in linearised order:
        for each edge (u, v) ∈ E:
            if dist(u) + l(u, v) < dist(v):
                dist(v) = dist(u) + l(u, v)
```

Complexity: Topological sort takes $\mathcal{O}(|V| + |E|)$. Processing takes $\mathcal{O}(|V| + |E|)$. Total: $\mathcal{O}(|V| + |E|)$ — faster than Dijkstra and handles negative weights!

DAG shortest paths

DAG (topological order: S, A, B, C, T): $S \to A$ (wt 1), $S \to B$ (wt 4), $A \to B$ (wt 2), $A \to C$ (wt 5), $B \to C$ (wt 1), $C \to T$ (wt 3).

Process	dist[S]	dist[A]	dist[B]	dist[C]	dist[T]
Init	0	∞	∞	∞	∞
S	0	1	4	∞	∞
A	0	1	3	6	∞
B	0	1	3	4	∞
C	0	1	3	4	7
T	0	1	3	4	7

Shortest distance from S to $T = 7$, via $S \to A \to B \to C \to T$.

Key Takeaway

On a DAG, process vertices in topological order and relax edges: $\mathcal{O}(|V| + |E|)$, handles negative weights.

3.10 Chapter Summary

Algorithm / Concept	**Complexity**	**Key Idea**
Connected components	$\mathcal{O}(\|V\| + \|E\|)$	DFS outer loop
Topological sort	$\mathcal{O}(\|V\| + \|E\|)$	Reverse DFS post order
Kosaraju's SCCs	$\mathcal{O}(\|V\| + \|E\|)$	DFS + reverse graph + DFS
BFS (unweighted)	$\mathcal{O}(\|V\| + \|E\|)$	Queue; level by level
Dijkstra (binary heap)	$\mathcal{O}((\|V\| + \|E\|) \log \|V\|)$	Greedy relaxation
DAG shortest paths	$\mathcal{O}(\|V\| + \|E\|)$	Topological order relaxation

3.11 Practice Problems

Level 1 — Recall and Understand

1. Define: connected component, DAG, topological order, SCC, weighted graph.
2. How does BFS differ from DFS? What does BFS compute that DFS does not?
3. What property of Dijkstra's algorithm makes it fail for negative edge weights?
4. What is "edge relaxation" in Dijkstra's algorithm?
5. Why can the DAG shortest path algorithm handle negative weights but Dijkstra cannot?
6. State the time complexity of: BFS, DFS, Dijkstra (with binary heap), Kosaraju, topological sort.

Level 2 — Apply

7. Run BFS from vertex A on the graph: A–B–C–D–A, B–D. Record the queue state after each step and the final distances.
8. Run Dijkstra from S on: $S \to A$ (wt 7), $S \to B$ (wt 3), $A \to C$ (wt 4), $B \to A$ (wt 1), $B \to C$ (wt 2), $C \to D$ (wt 5). Show the distance table after each extraction.
9. Find the topological sort of: $V = \{A, B, C, D, E\}$, $E = \{A{\to}C, B{\to}C, C{\to}D, C{\to}E\}$.

10. Apply Kosaraju's algorithm to find SCCs of: 1→2, 2→3, 3→1, 3→4, 4→5, 5→3.
11. Find the shortest path from S to all vertices in the DAG: $S \to A$ (1), $S \to B$ (5), $A \to B$ (2), $A \to C$ (4), $B \to C$ (1), $C \to T$ (3). What is the shortest path to T?

Level 3 — Prove

12. Prove that BFS computes correct shortest distances in an unweighted graph.
13. Prove: a directed graph has a topological ordering if and only if it is a DAG.
14. Prove: when Dijkstra extracts vertex u, dist$[u]$ is the true shortest distance from s to u (assuming non-negative weights).

Level 4 — Challenge

15. Modify BFS to work on a directed graph and determine reachability from a given source.
16. Give an example of a directed graph with a negative cycle (a cycle whose total edge weight is negative). Why does Dijkstra fail on such a graph?

Chapter 4

Greedy Algorithms and Dynamic Programming

Module IV · 12 Hours · Sections 5.1, 5.4, 6.1, 6.6 of Dasgupta et al.

What you will learn in this chapter.

- The greedy strategy: always make the locally best choice.
- Minimum spanning trees (MSTs) and the Cut Property theorem.
- Kruskal's algorithm: build an MST by sorting edges globally.
- The Union-Find data structure: track which vertices are in which component.
- Prim's algorithm: build an MST by growing a single tree.
- Dynamic programming: solving problems by combining overlapping sub-solutions.
- Floyd-Warshall algorithm: all-pairs shortest paths in $\mathcal{O}(|V|^3)$.

4.1 The Greedy Strategy

A **greedy algorithm** makes decisions step by step, always choosing the option that looks best at the current moment, without reconsidering previous choices.

Intuition

Imagine you are paying for a purchase of Rs. 87 and have coins of denominations Rs. 50, 20, 10, 5, and 1. A greedy strategy: always use the largest coin that fits. $50 + 20 + 10 + 5 + 1 + 1 = 87$. Done with 6 coins.
For standard coin systems, this greedy approach gives the minimum number of coins. But greedy does not always give the optimal answer! (Consider coins: 1, 3, 4. Target: 6. Greedy: 4+1+1 = 3 coins; optimal: 3+3 = 2 coins.)
The key question for any greedy algorithm: does the locally optimal choice always lead to a globally optimal solution?

For minimum spanning trees, the answer is yes, as we prove via the Cut Property.

4.2 Minimum Spanning Trees

4.2.1 The Problem

Definition

Given a connected, weighted, undirected graph $G = (V, E, w)$, a **spanning tree** is a subgraph that:

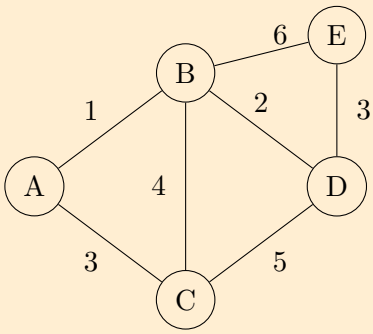
- is a tree (connected and acyclic), and
- spans all vertices (includes every vertex in V).

A **minimum spanning tree (MST)** is a spanning tree of minimum total edge weight.

Intuition

An internet service provider wants to connect 10 towns with fibre-optic cable. Each pair of towns has a different cable-laying cost. They want to connect all towns while minimising total cost. This is exactly the MST problem: find the spanning tree of minimum weight.
A spanning tree with n vertices has exactly $n - 1$ edges. We want those $n - 1$ edges to have minimum total weight.

A graph and its MST



MST edges (one possible set): $\{AB, BD, DE, AC\}$ with total weight $1+2+3+3 = 9$.

4.2.2 The Cut Property: Foundation of MST Algorithms

Definition

A **cut** of a graph $G = (V, E)$ is a partition of V into two non-empty sets S and $V \setminus S$. An edge **crosses the cut** if one endpoint is in S and the other is in $V \setminus S$.

Theorem 4.1 (Cut Property). *Let e be the unique minimum-weight edge crossing some cut $(S, V \setminus S)$. Then e is in every MST of G.*

Intuition

Proof idea. Suppose for contradiction that some MST T does not contain e. Adding e to T creates a cycle. This cycle must contain at least one other edge e' crossing the cut. Since e is the unique minimum crossing edge, $w(e) < w(e')$. Replacing e' with e gives a spanning tree of smaller weight — contradicting T being an MST.

The Cut Property is the theoretical justification for both Kruskal's and Prim's algorithms. They both construct MSTs by repeatedly selecting the minimum-weight edge that crosses some cut.

4.3 Kruskal's Algorithm

4.3.1 The Idea

Kruskal's algorithm takes a *global* view: sort all edges by weight, then add them one by one (cheapest first), skipping any edge that would create a cycle.

Algorithm: Kruskal(G, w)

```
Kruskal(G, w):
    for all u ∈ V: makeset(u)
    X = ∅
    Sort all edges by weight w(u, v)
    for each edge (u, v) in increasing order of weight:
        if find(u) ≠ find(v):
            add (u, v) to X
            union(u, v)
    return X
```

Intuition

Think of building a railway network cheaply. You have a list of all possible railway lines, sorted by construction cost. You keep building the cheapest line that connects two previously unconnected towns (a line within an already-connected town-cluster would create a cycle and is useless).

Kruskal's on a 5-vertex graph

Edges sorted by weight: $AB(1), BD(2), AC(3), DE(3), BC(4), CD(5), BE(6)$.

Edge	Weight	Creates cycle?	Action	Components
AB	1	No	Add	$\{A,B\}, \{C\}, \{D\}, \{E\}$
BD	2	No	Add	$\{A,B,D\}, \{C\}, \{E\}$
AC	3	No	Add	$\{A,B,C,D\}, \{E\}$
DE	3	No	Add	$\{A,B,C,D,E\}$
BC	4	**Yes**	Skip	(all 5 vertices connected)

MST $= \{AB, BD, AC, DE\}$, total weight $= 1+2+3+3 = 9$.

Complexity: Sorting $|E|$ edges: $\mathcal{O}(|E| \log |E|)$. Cycle detection (using Union-Find, see next section): nearly $\mathcal{O}(|E|)$. Total: $\mathcal{O}(|E| \log |E|)$.

Key Takeaway

Kruskal's algorithm greedily adds the cheapest safe edge (no cycle) until the MST is complete.

4.4 Union-Find: A Data Structure for Disjoint Sets

Kruskal's algorithm needs to quickly answer: "Do u and v belong to the same connected component of the current partial MST?"

This is the **Union-Find** (or **disjoint set union**) problem.

4.4.1 The Operations

We maintain a collection of disjoint sets. We need:

- FIND(x): return the "representative" (root) of the set containing x.
- UNION(x, y): merge the sets containing x and y.

Two vertices u and v are in the same component iff FIND(u) = FIND(v).

4.4.2 Implementation: Forest with Rank and Path Compression

Represent each set as a *tree*. Each element points to its parent; the root points to itself. FIND(x): walk up to the root. UNION(x, y): attach the root of one tree under the root of the other.

Two optimisations make this blazingly fast:

1. **Union by Rank:** Always attach the shorter tree under the taller tree. This keeps trees shallow.
2. **Path Compression:** When doing FIND(x), make every node on the path to the root point directly to the root. This flattens the tree for future queries.

Algorithm: find(x) with path compression

```
find(x):
    if π(x) ≠ x: π(x) = find(π(x)) [π(x) is the parent of
        x]
    return π(x)
```

Algorithm: union(x, y) by rank

```
union(x, y):
    rx = find(x); ry = find(y)
    if rank(rx) > rank(ry): π(ry) = rx
    else: π(rx) = ry
    if rank(rx) = rank(ry): rank(ry) = rank(ry) + 1
```

Union-Find trace

Initial sets: $\{A\}, \{B\}, \{C\}, \{D\}, \{E\}$ (each is its own root).

1. UNION(A, B): merge. Root of one (say A) becomes parent. Sets: $\{A, B\}, \{C\}, \{D\}, \{E\}$.
2. UNION(C, D): Sets: $\{A, B\}, \{C, D\}, \{E\}$.
3. UNION(A, C): merge $\{A, B\}$ and $\{C, D\}$. Root of A's tree points to root of C's tree (or vice versa, depending on rank). Sets: $\{A, B, C, D\}, \{E\}$.
4. FIND(B): follow $B \to A \to$ root. With path compression: B now points directly to root.
5. UNION(A, E): Sets: $\{A, B, C, D, E\}$.

Theorem 4.2. *With union by rank and path compression, a sequence of m operations on n elements runs in $\mathcal{O}(m \cdot \alpha(n))$ time, where $\alpha(n)$ is the* ***inverse Ackermann function*** *— a function that is effectively constant (at most 4) for any n that could ever arise in practice.*

Intuition

$\alpha(n) \leq 4$ for any $n \leq 10^{80}$ (the number of atoms in the observable universe). So Union-Find with both optimisations is, for all practical purposes, $\mathcal{O}(1)$ per operation.

Exam Tip

For Kruskal's questions:

1. Sort edges by weight.
2. Use Union-Find to check (and update) components: add edge (u, v) iff FIND$(u) \neq$ FIND(v).
3. Continue until $|V| - 1$ edges are added.

You may be asked to trace Union-Find operations separately.

4.5 Prim's Algorithm

4.5.1 The Idea

Prim's algorithm takes a *local* view: start from a single vertex, and repeatedly grow the MST by adding the cheapest edge that connects a tree vertex to a non-tree vertex.

Intuition

Imagine building a road network starting from one city. At each step, build the cheapest road that connects an already-connected city to a new city. Never connect two already-connected cities (that would be redundant).

Algorithm: Prim(G, w, s)

```
Prim(G, w, s):
    for all u ∈ V: cost(u) = ∞, prev(u) = nil
    cost(s) = 0
    H = makequeue(V)    [priority queue keyed by cost]
    while H is not empty:
        v = deletemin(H)
        for each edge (v, u) ∈ E:
            if cost(u) > w(v, u):
                cost(u) = w(v, u)
                prev(u) = v
                decreasekey(H, u)
```

Prim's algorithm starting from A

Graph: A–$B(1)$, A–$C(3)$, B–$D(2)$, B–$C(4)$, C–$D(5)$, D–$E(3)$, B–$E(6)$.

Step	Extract	cost[B]	cost[C]	cost[D]	cost[E]
Init	—	∞	∞	∞	∞
From A	A	**1** (via A)	**3** (via A)	∞	∞
From B	B (cost 1)	1	3	**2** (via B)	**6** (via B)
From D	D (cost 2)	1	3	2	**3** (via D)
From C	C (cost 3)	1	3	2	3
From E	E (cost 3)	1	3	2	3

MST edges added: AB, BD, AC, DE. Total weight: $1+2+3+3=9$. ✓

Complexity: With a binary heap: $\mathcal{O}((|V|+|E|)\log|V|)$. With a simple array: $\mathcal{O}(|V|^2)$ (better for dense graphs).

4.5.2 Kruskal vs. Prim

Feature	**Kruskal**	**Prim**
Approach	Global: sort all edges first	Local: grow from one vertex
Data structure	Union-Find	Priority queue
Best for	Sparse graphs ($\|E\|$ small)	Dense graphs ($\|E\|$ large)
Time complexity	$\mathcal{O}(\|E\|\log\|E\|)$	$\mathcal{O}((\|V\|+\|E\|)\log\|V\|)$ with heap $\mathcal{O}(\|V\|^2)$ with array

Exam Tip

Both algorithms are exam-ready. For a given graph, you may be asked to:

1. Apply Kruskal's (sort edges, add greedily).
2. Apply Prim's (grow from a given source, track min costs).
3. State and apply the Cut Property to justify why a specific edge must be in the MST.

Key Takeaway

Prim grows the MST one vertex at a time (like Dijkstra but minimises edge weight, not path length); Kruskal processes all edges globally sorted by weight.

4.6 Introduction to Dynamic Programming

4.6.1 What Is Dynamic Programming?

Intuition

In Chapter 1, we saw that naive recursive Fibonacci was slow because it recomputed the same sub-answers thousands of times. Memoisation fixed this by storing sub-answers.

Dynamic programming (DP) is the systematic, table-filling version of memoisation. Instead of computing answers on demand (top-down), DP fills a table bottom-up, using previously computed entries to compute new ones.

The two hallmarks of a problem solvable by DP:

1. **Optimal substructure:** the optimal solution to the big problem contains optimal solutions to sub-problems.
2. **Overlapping sub-problems:** the same sub-problems appear multiple times (so caching pays off).

Definition

- **Optimal substructure**: an optimal solution to the full problem can be constructed from optimal solutions to its sub-problems.
- **Overlapping sub-problems**: the recursive solution to the full problem reuses the same sub-problem solutions multiple times.

A problem with both properties is a candidate for dynamic programming.

4.6.2 DP on DAGs: Shortest Paths Revisited

The DAG shortest path algorithm from Section 3.9 is actually a DP. Let $\text{dist}(v) =$ shortest distance from source s to v.

$$\text{dist}(v) = \min_{(u,v)\in E} \Big[\text{dist}(u) + w(u,v)\Big]$$

This defines $\text{dist}(v)$ in terms of $\text{dist}(u)$ for predecessors u. If we fill this table in topological order, each entry is computed exactly once.

Intuition

The recurrence above is a DP formula. The topological order ensures that when we compute $\text{dist}(v)$, all values $\text{dist}(u)$ (for predecessors u) are already in the table.

4.7 All-Pairs Shortest Paths: Floyd-Warshall

Dijkstra finds shortest paths from *one* source. If we want shortest paths between *every pair* of vertices, we could run Dijkstra $|V|$ times — but Floyd-Warshall gives a simpler, elegant DP solution.

4.7.1 The Problem

Given a weighted directed graph $G = (V, E, w)$ (possibly with negative edge weights, but no negative cycles), find the shortest path distance $d(i, j)$ for every pair of vertices i, j.

4.7.2 The DP Formulation

Let $d_k(i, j)$ = the length of the shortest path from i to j using only vertices $\{1, 2, \ldots, k\}$ as intermediate vertices.

Base case ($k = 0$): no intermediate vertices.

$$d_0(i,j) = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if there is an edge } (i,j) \\ \infty & \text{otherwise} \end{cases}$$

Recurrence: to use vertex k, either avoid k (same as $d_{k-1}(i, j)$) or route through k:

$$d_k(i,j) = \min\Big(d_{k-1}(i,j),\ d_{k-1}(i,k) + d_{k-1}(k,j)\Big)$$

Intuition

Think of it this way: "Can I improve the path from i to j by going through vertex k?" If yes, the new distance is (distance i to k) + (distance k to j). If no, keep the old distance. We do this for $k = 1, 2, \ldots, |V|$, updating the distance matrix each time.

Algorithm: Floyd-Warshall(G)

```
Floyd-Warshall(G):
    [Initialise: dist[i][i] = 0; dist[i][j] = w(i,j) for
       edges; ∞ otherwise]
    for k = 1 to n:
        for i = 1 to n:
            for j = 1 to n:
                dist[i][j] = min(dist[i][j], dist[i][k] +
                    dist[k][j])
```

Complexity: Three nested loops of length $|V|$: $\mathcal{O}(|V|^3)$.

Note: in practice, we use a single matrix D and update in place (since $D[i][k]$ and $D[k][j]$ do not change when k is fixed).

Floyd-Warshall on a 4-vertex graph

Graph: $1 \to 2$ (wt 3), $1 \to 3$ (wt 8), $2 \to 4$ (wt 1), $3 \to 2$ (wt 4), $4 \to 1$ (wt 2), $4 \to 3$ (wt -5).

Initial matrix $D^{(0)}$ (∞ = no direct edge):

	1	2	3	4
1	0	3	8	∞
2	∞	0	∞	1
3	∞	4	0	∞
4	2	∞	-5	0

After $k = 1$ (routing through vertex 1): Only vertex 4 has a finite path to vertex 1 ($D[4][1] = 2$). $D[4][2]$: $\min(\infty,\ 2{+}3) = 5$. Updated! $D[4][3]$: $\min(-5,\ 2{+}8) = -5$. No change. (All other rows: $D[i][1] = \infty$, so no updates.)

After $k = 2$ (routing through vertex 2): $D[1][4]$: $\min(\infty,\ 3 + 1) = 4$. Updated! $D[3][4]$: $\min(\infty,\ 4 + 1) = 5$. Updated!

After $k = 3$ (routing through vertex 3): $D[4][2]$: $\min(5,\ -5 + 4) = -1$. Updated!

After $k = 4$ (routing through vertex 4) — **final matrix**:

	1	2	3	4
1	0	3	-1	4
2	3	0	-4	1
3	7	4	0	5
4	2	-1	-5	0

Reading the result: $D[1][3] = -1$. Shortest path from 1 to 3: $1 \to 2 \to 4 \to 3$, weight $= 3 + 1 + (-5) = -1$. ✓
$D[3][1] = 7$: path $3 \to 2 \to 4 \to 1$, weight $= 4 + 1 + 2 = 7$. ✓

Exam Tip

Floyd-Warshall is a common exam question. Know how to:

- Set up the initial distance matrix from a graph.
- Apply one iteration of the recurrence for a given k.
- State the time complexity: $\mathcal{O}(|V|^3)$.
- State when Floyd-Warshall works: any graph without negative cycles.
- Detect a negative cycle: if $D[i][i] < 0$ after the algorithm, there is a negative cycle containing i.

Key Takeaway

Floyd-Warshall fills a $|V| \times |V|$ distance matrix in $\mathcal{O}(|V|^3)$ time by checking, for each pair (i, j), whether routing through vertex k improves the known distance.

4.8 Chapter Summary

Algorithm / Concept	Complexity	Key Idea
MST (definition)	—	Spanning tree of min weight
Cut Property	—	Min crossing edge is in every MST
Kruskal's algorithm	$\mathcal{O}(\|E\| \log \|E\|)$	Sort edges, greedily add
Union-Find (rank + path comp.)	$\mathcal{O}(\alpha(n))$ per op.	Nearly constant; disjoint sets
Prim's algorithm (heap)	$\mathcal{O}((\|V\| + \|E\|) \log \|V\|)$	Grow tree from source
Prim's algorithm (array)	$\mathcal{O}(\|V\|^2)$	Better for dense graphs
Dynamic programming	—	Optimal substructure + overlapping sub-proble
Floyd-Warshall	$\mathcal{O}(\|V\|^3)$	DP on intermediate vertices

4.9 Practice Problems

Level 1 — Recall and Understand

1. Define: spanning tree, minimum spanning tree, cut, edge crossing a cut.
2. State the Cut Property for MSTs. Why does it justify greedy MST algorithms?
3. What is the difference between Kruskal's and Prim's algorithms?
4. What are the two operations supported by Union-Find? What is the purpose of union by rank and path compression?
5. What are the two conditions a problem must satisfy to be solvable by dynamic programming?
6. State the Floyd-Warshall recurrence. What does $d_k(i, j)$ represent?
7. What is the time complexity of Floyd-Warshall? What is it used for?
8. Under what condition does Floyd-Warshall give wrong answers?

Level 2 — Apply

9. Apply Kruskal's algorithm to the graph: $\{AB(4), AC(2), AD(7), BC(3), BD(6), CD(5)\}$. Show the sorted edge list, which edges are added, and the final MST.
10. Apply Prim's algorithm starting from vertex A on the same graph as above. Show the cost table after each extraction and the MST.
11. Trace Union-Find operations for Kruskal's algorithm on Problem 9: after each edge addition, show the parent/root of each vertex.
12. Apply Floyd-Warshall to the graph: vertices $\{1, 2, 3\}$, edges 1→2 (wt 4), 2→3 (wt 3), 1→3 (wt 10). Write the initial matrix $D^{(0)}$ and the final matrix $D^{(3)}$.
13. In the Floyd-Warshall algorithm for a 4-vertex graph, write out the update for $D[1][3]$ when $k = 2$.
14. Identify which of the following problems have optimal substructure and overlapping subproblems: (a) Fibonacci numbers; (b) Binary search; (c) Merge sort; (d) Shortest path in a DAG.

Level 3 — Prove

15. Prove the Cut Property: the minimum-weight edge crossing any cut belongs to every MST.
16. Prove that an MST of a graph with n vertices has exactly $n - 1$ edges.

17. Prove that Kruskal's algorithm produces an MST. (Hint: use the Cut Property and consider the first edge Kruskal selects that differs from a given MST T^*.)
18. Prove the correctness of Floyd-Warshall: at the end of iteration k, $d_k(i, j)$ equals the shortest path from i to j using only vertices from $\{1, \ldots, k\}$ as intermediates.

Level 4 — Challenge

19. A graph has $|V|$ vertices. How many different spanning trees can it have? (Give a small example.) Does Kruskal's algorithm always produce the same MST? Under what conditions might two different MSTs exist?
20. Modify the Floyd-Warshall algorithm to also compute the actual shortest path (not just the distance) between every pair of vertices. Describe the additional information you would store.

Quick Reference: Complexity Summary

The table below lists every algorithm covered in Modules I–IV with its time complexity. Use this for quick revision before the exam.

Algorithm	**Module**	**Time Complexity**	**Notes**
Fibonacci (recursive)	I	$\mathcal{O}(\varphi^n)$	Exponential
Fibonacci (memoised)	I	$\mathcal{O}(n)$	Top-down DP
Fibonacci (iterative)	I	$\mathcal{O}(n)$	Space: $\mathcal{O}(1)$
Integer addition	I	$\mathcal{O}(n)$	n = number of bits
Integer multiplication	I	$\mathcal{O}(n^2)$	Grade-school method
Euclid's GCD	I	$\mathcal{O}(\log \min(a, b))$	In number of steps
Modular exponentiation	I	$\mathcal{O}((\log n)^2)$	Repeated squaring
Trial division (primality)	I	$\mathcal{O}(\sqrt{n})$	Exponential in bits
Fermat primality test	I	$\mathcal{O}(k \log^2 n)$	k = number of trials
Karatsuba multiplication	II	$\mathcal{O}(n^{1.585})$	Divide and conquer
Binary search	II	$\mathcal{O}(\log n)$	Sorted array
Merge sort	II	$\mathcal{O}(n \log n)$	Stable sort
DFS	II	$\mathcal{O}(\|V\| + \|E\|)$	All vertices and edges
BFS	III	$\mathcal{O}(\|V\| + \|E\|)$	Shortest hop-paths
Kosaraju (SCCs)	III	$\mathcal{O}(\|V\| + \|E\|)$	Two DFS passes
Topological sort	III	$\mathcal{O}(\|V\| + \|E\|)$	DAGs only
Dijkstra (binary heap)	III	$\mathcal{O}((\|V\| + \|E\|) \log \|V\|)$	Non-negative weights
Dijkstra (array)	III	$\mathcal{O}(\|V\|^2)$	Dense graphs
DAG shortest paths	III	$\mathcal{O}(\|V\| + \|E\|)$	Allows neg. weights
Kruskal's MST	IV	$\mathcal{O}(\|E\| \log \|E\|)$	Sort edges
Prim's MST (heap)	IV	$\mathcal{O}((\|V\| + \|E\|) \log \|V\|)$	Grow tree
Union-Find (rank+path)	IV	$\mathcal{O}(\alpha(n))$ per op.	Nearly constant
Floyd-Warshall	IV	$\mathcal{O}(\|V\|^3)$	All-pairs paths