



CONTENTS

1	Other Graph Algorithms and Proofs	5
1.1	Adjacency Matrices	5
1.2	Prim's Algorithm for Minimum Spanning Trees	8
1.3	A* (A-star) Search Algorithm	8
1.3.1	The Path Cost $g(n)$	9
1.3.2	The Heuristic Function $h(n)$	9
1.3.3	The Total Estimated Distance $f(n)$ and Priority Queues	10
1.4	Injection, Surjection, Bijection, and Isomorphism	11
1.5	Travelling Salesperson Problem and Nearest Neighbour	14
1.6	Topological Sort and DAGs	15
1.7	Proofs	17
1.7.1	Example 1: Bipartite	17
1.7.2	Example 2 and 3. Cycle Detection and Leaves	19
2	How the Internet Works	21
2.1	A Network of Networks	21
2.2	Packets	21
2.2.1	TCP: Getting It All, in Order	22
2.2.2	UDP: Fast, No Promises	23
2.3	IP Addresses	23
2.3.1	Networks Have Prefixes	24
2.3.2	Ports	24
2.4	Routing: Shortest Paths for Packets	25
2.4.1	Two Ways to Share the Map	27

2.4.2	The Speed of Light Is a Speed Limit	27
2.5	DNS: The Internet's Phone Book	28
2.6	Putting It Together	28
3	HTML	33
3.1	HTML Elements	33
3.2	Attributes	34
3.3	HTML Document Structure	34
3.4	Web Development and HTML	36
3.5	Metadata	37
3.6	Div Elements	38
3.7	Semantic HTML	39
3.8	Headings and Sections	41
3.9	Text Elements	43
3.10	Links	45
3.11	Lists	45
3.11.1	Unordered Lists	46
3.11.2	Ordered Lists	46
3.12	Navigation	49
3.13	Tables	50
3.14	Forms	54
3.15	Images	56
3.16	The Finished Page	59
4	Web APIs	63
4.1	Making Web Requests with HTTP	63
4.1.1	HTTP Requests	63
4.1.2	Using HTTP with Web-Based APIs	64
4.1.3	HTTP Responses	65
4.1.4	Data Formats in Responses	66
4.2	Calling a Real API from Python	66
5	Building a Web API	71
5.1	Setting up an API	71
5.1.1	What is an endpoint?	72
5.1.2	Reading all resources	73
5.2	Requests with cURL	74
5.3	Creating resources with POST	75
5.4	Updating and Deleting Resources with cURL	77

5.4.1	Fully replacing a resource with PUT	77
5.4.2	Partially updating a resource with PATCH	78
5.4.3	Removing a resource with DELETE	79
5.4.4	Headers with -i and -v	79
5.5	Authentication and API Keys	80
5.6	Postman	82
5.6.1	Building your first request in Postman	82
5.6.2	Why this matters once requests get complicated	83
5.7	More Practice	84
5.8	Summary	85
6	CSS and JavaScript	87
6.1	Setting Up	87
6.2	Your First Stylesheet	88
6.3	Selectors and the Cascade	89
6.4	The Box Model	90
6.5	Layout with Flexbox	91
6.6	JavaScript for Python Programmers	92
6.7	Making the Page Respond	93
6.7.1	Dark Mode	93
6.7.2	Crossing Off Steps	95
6.7.3	Doubling the Recipe	95
6.8	Talking to Your Server	97
6.8.1	Serving the Page with Express	98
6.8.2	The Like Button	99
6.9	Showing Your Friends	99
7	Digital Logic: From Transistors to Adders	105
7.1	Bits as Voltages	105
7.2	Transistors Are Switches	105
7.3	The NAND Gate	106
7.4	The Gate Family	107
7.5	NAND Can Build Anything	108
7.6	Adding in Binary	109
7.6.1	The Half Adder	109
7.6.2	The Full Adder	109
7.6.3	The Ripple-Carry Adder	110
7.7	Checking the Design in Python	110
7.8	Subtraction for Free	112

7.9 From Adders to Computers	112
A Answers to Exercises	117
Index	133

Other Graph Algorithms and Proofs

How does a power company wire a town with the least cable? How does a video game character find its way around walls? How does a build system know which tasks can run at the same time? Each is a graph problem. This chapter is a tour: adjacency matrices, Prim's algorithm for minimum spanning trees, A* search, isomorphism, the Traveling Salesperson Problem, topological sorting, and some classic proofs. Several of these ideas are advanced; you will meet them again in college computer science.

1.1 Adjacency Matrices

Recall that we can represent graphs as lists of data called adjacency lists. In code, this could look like the following:

- A: [B, C]
- B: [A, D]
- C: [A]
- D: [B]

This represents the following graph

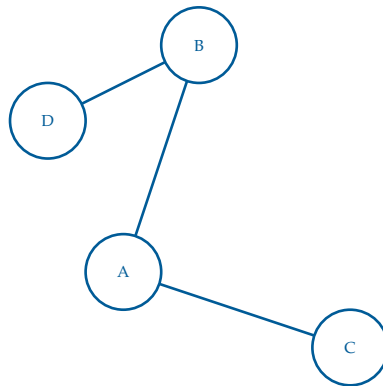


Figure 1.1: A simple (undirected unweighted) graph used to make an adjacency list and adjacency matrix.

We could also represent the same data as a *matrix*, which we have covered in linear algebra. The *adjacency matrix* is 2D array (as compared to an adjacency list, which consists of multiple 1D arrays) where $\text{matrix}[i][j]$ indicates whether an edge exists from vertex i to vertex j . Commonly, $\text{matrix}[i][j] = 1$ if and only if the edge from vertex i to vertex j exists, else $\text{matrix}[i][j] = 0$.¹The above graph in Figure 1.1 can be represented as:

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

For *undirected* graphs, the adjacency matrix is symmetric: it equals its own transpose.

$$A = A^T$$

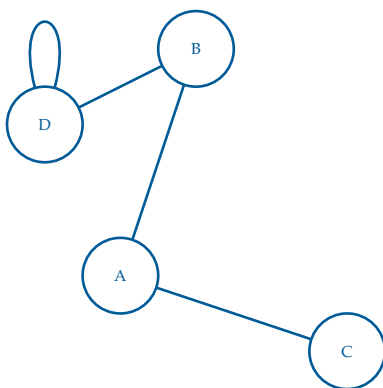


Figure 1.2: A simple graph with an added loop on vertex D.

By convention, a diagonal entry such as $\text{matrix}[k][k]$, where $(k \in \mathbb{Z})$, is non-zero only if there is a loop from a vertex to itself. The value of the entry represents the number of such loops. In this case, Figure 1.2 shows one loop on vertex D, so the diagonal entry is 1. (Some texts count an undirected loop twice and use 2.)

For *directed* graphs, the adjacency matrix looks a bit different. A *directed adjacency matrix* represents a directed graph, where edges have a direction (arrows). When $A_{ij} = 1$, then an edge exists from vertex i to vertex j .

¹If the edges are weighted, the $\text{matrix}[i][j]$ is equal to the weight, or 0 if the edge does not exist.

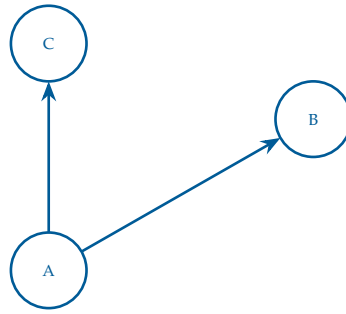


Figure 1.3: A simple directed graph used to make an adjacency matrix.

The resulting adjacency matrix is

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Notice how, now, $A \neq A^T$, because the edges are directional.

Exercise 1 Adjacency Matrix Exercise

Working Space

The following is an adjacency matrix for a weighted (undirected) graph.

	A	B	C	D	E	F	G
A	0	4	2	0	0	0	0
B	4	0	1	5	0	0	0
C	2	1	0	8	10	0	0
D	0	5	8	0	2	6	0
E	0	0	10	2	0	3	7
F	0	0	0	6	3	0	4
G	0	0	0	0	7	4	0

Draw the corresponding graph with weighted edges.

Answer on Page 117

1.2 Prim's Algorithm for Minimum Spanning Trees

A *spanning tree* of a connected graph is a tree that includes every vertex, using some of the edges. A *minimum spanning tree* (MST) is a spanning tree whose edge weights add up to the smallest possible total. MSTs are used to connect cities, computer networks, or power grids with the least amount of cable.

Prim's Algorithm builds the MST one edge at a time:

1. Start with any vertex in the tree.
2. Among all edges that connect the tree to a vertex outside it, choose the cheapest.
3. Add that edge and its new vertex to the tree.
4. Repeat until every vertex is in the tree.

Worked example. Run Prim's algorithm on the weighted graph from the adjacency matrix exercise, starting at A. The cheapest edge out of {A} is A–C (2), so add C. Out of {A, C}, the cheapest edge is C–B (1), so add B. Out of {A, B, C}, the cheapest is B–D (5), so add D. Out of {A, B, C, D}, the cheapest is D–E (2), so add E. Then E–F (3) is cheapest, so add F. Last, F–G (4) is cheaper than E–G (7), so add G. The tree edges are A–C, C–B, B–D, D–E, E–F, and F–G. Their total weight is $2 + 1 + 5 + 2 + 3 + 4 = 17$.

Exercise 2 Prim's Algorithm from G

Use the same weighted graph from the adjacency matrix exercise. Start at G and list the edges in the order Prim's algorithm adds them. What is the total weight of the minimum spanning tree?

Working Space

Answer on Page 117

1.3 A* (A-star) Search Algorithm

The last shortest path we will discuss in this chapter is the *A-star Algorithm*, more commonly represented as *A* Algorithm*. In essence, it is an extension of Dijkstra's Algorithm in combination with a *priority-based heuristic function*. A heuristic is an introduced estimated cost from any node to the goal.

The key behind this implementation is the following function

$$f(n) = g(n) + h(n)$$

where

- $g(n)$: the cost of the path from the start node (usually S) to the current node n
- $h(n)$: the *heuristic* function that estimates the remaining cost from n to the goal (usually E)
- $f(n) = g(n) + h(n)$: the estimated total cost of a path from S to E passing through node n

1.3.1 The Path Cost $g(n)$

$g(n)$ remains to be the same value that we used in Dijkstra's and other shortest path algorithms.

When moving from a node u to an adjacent node v , the path cost is *updated* as

$$g(v) = g(u) + c(u, v)$$

where $c(u, v)$ is the cost of traversing the edge from u to v . As we discussed in the chapter on Dijkstra's, we have a decision to make upon each visit to a node:

$$\text{if } \text{dist}[u] + w(u, v) < \text{dist}[v] \text{ then update } \text{dist}[v]$$

1.3.2 The Heuristic Function $h(n)$

The heuristic $h(n)$ function depends on the problem domain. Its purpose is to estimate the remaining cost from node n to the goal. While this is a difficult task, we can use basic algebra to make an informed guess.

For example, in a map-based pathfinding problem, $h(n)$ may be the straight-line distance, or *Euclidean distance* between node n and the goal. Since the actual path cannot be shorter than the Euclidean distance, this provides a reasonable estimate of the remaining cost. If a node's location is represented on a 2D plane (such as a map or graph) (x_n, y_n) and the goal location as (x_g, y_g) , the Euclidean Heuristic

$$h(n) = \sqrt{(x_g - x_n)^2 + (y_g - y_n)^2}$$

Many 2D games instead use *taxicab distance* (for 4-way movement),

$$h(n) = |x_g - x_n| + |y_g - y_n|$$

or *Chebyshev distance* (typically for 8-way movement in a game)

$$h(n) = \max(|x_g - x_n|, |y_g - y_n|)$$

which takes the maximum of the horizontal and vertical movements.

Sometimes, graphs without direct spatial meaning become more difficult to work with because there is no obvious notion of distance between two nodes.

For general graphs, such as social networks, communication networks, or dependency graphs, the heuristic must be derived from problem-specific knowledge. For example, a social network application might use information such as mutual connections, community membership, or other metadata to estimate how *close* a user is to a target user.

In many cases, however, no suitable heuristic is known. In these situations, the heuristic is often defined as

$$h(n) = 0$$

for all nodes n . Substituting this into the A^* evaluation function gives

$$f(n) = g(n)$$

which causes A^* to reduce to Dijkstra's algorithm. While this guarantees an optimal path, it loses the efficiency gains that a good heuristic can provide.

There is an important property that the Heuristic Function must satisfy:

$$h(n) \leq \text{true remaining cost}$$

When an admissible heuristic is used (such that $h(n) \leq \text{true remaining cost}$), A^* is guaranteed to find an optimal path if one exists.

1.3.3 The Total Estimated Distance $f(n)$ and Priority Queues

The purpose of the function

$$f(n) = g(n) + h(n)$$

is to assign a priority to each node during the search. Recall from our discussion of priority queues that each element is associated with a priority value, and the element with the

highest (or lowest) priority is removed first.

In A^* , the priority of a node is its estimated total distance $f(n)$. Nodes with smaller values of $f(n)$ are considered more promising because they represent paths that appear likely to reach the goal with a lower total cost.

As with Dijkstra's algorithm, A^* maintains a priority queue containing nodes that have been discovered but not yet fully explored. However, instead of prioritizing nodes solely by their known distance from the start node, A^* prioritizes nodes according to their estimated total distance

$$\text{priority}(n) = f(n)$$

At each step, the node with the smallest $f(n)$ value is removed from the priority queue and explored. When a neighboring node is discovered, its path cost g is updated, a new heuristic estimate $h(n)$ is calculated, and a new value of $f(n)$ is computed.

There are some good references about this in your digital resources, as well as a fun Minecraft-based explanation [here](#).

1.4 Injection, Surjection, Bijection, and Isomorphism

Let us talk about Graph Isomorphism. To do this, we must first introduce the terms of *injection*, *surjection*, and *bijection*. These are types of functions in set theory as well as mathematics.

Definition 1 (Injection). A function $f : A \rightarrow B$ is *injective* if different inputs always produce different outputs. Importantly, this means no two inputs map to the same output.

Definition 2 (Surjection). A function $f : A \rightarrow B$ is *surjective* if every element of B is hit by at least one element of A . In essence, all of the elements of B are covered.

Definition 3 (Bijection). A function $f : A \rightarrow B$ is *bijective* if it is both injective and surjective. Ultimately, the function creates no collisions and hits every element of B .

Exercise 3 Bijjective, Surjective, Injective Exercise

Determine whether each function is injective, surjective, bijective, or none of these.

Working Space

1. $f : \{1, 2, 3\} \rightarrow \{a, b, c, d\}$ defined by

$$f(1) = a, \quad f(2) = b, \quad f(3) = c.$$

2. $g : \{1, 2, 3, 4\} \rightarrow \{a, b, c\}$ defined by

$$g(1) = a, \quad g(2) = b, \quad g(3) = c, \quad g(4) = a.$$

3. $h : \{1, 2, 3\} \rightarrow \{a, b, c\}$ defined by

$$h(1) = b, \quad h(2) = c, \quad h(3) = a.$$

Answer on Page 118

Now, let us talk about *Isomorphism*. A *graph isomorphism* is a bijection between the vertex sets of two graphs that preserves adjacency. Thus, an isomorphism not only matches vertices one-to-one, but also preserves the structure of the graph.

Definition 4 (Graph Isomorphism). Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be graphs. An *isomorphism from G to H* is a bijection

$$f : V_G \rightarrow V_H$$

such that for all vertices $u, v \in V_G$,

$$\{u, v\} \in E_G \iff \{f(u), f(v)\} \in E_H.$$

If such a function exists, we say that G and H are *isomorphic* and write

$$G \cong H.$$

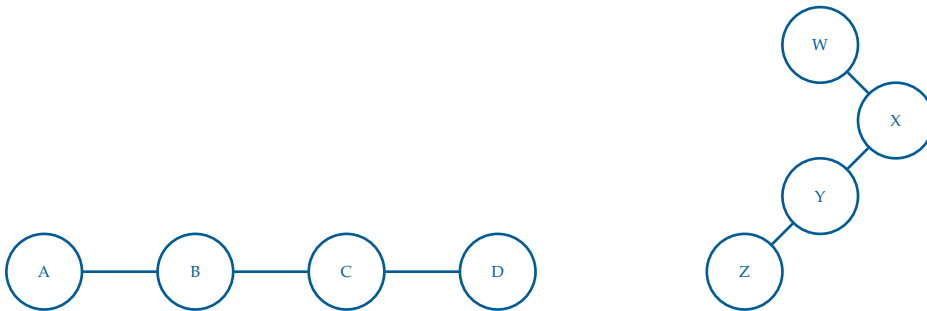


Figure 1.4: Two graphs that are isomorphic. Graph G (left) and Graph H (right).

We are able to say that these graphs (Figure 1.4) are isomorphic as one vertex from G can be paired with one vertex from H , while maintaining the same edge connections.

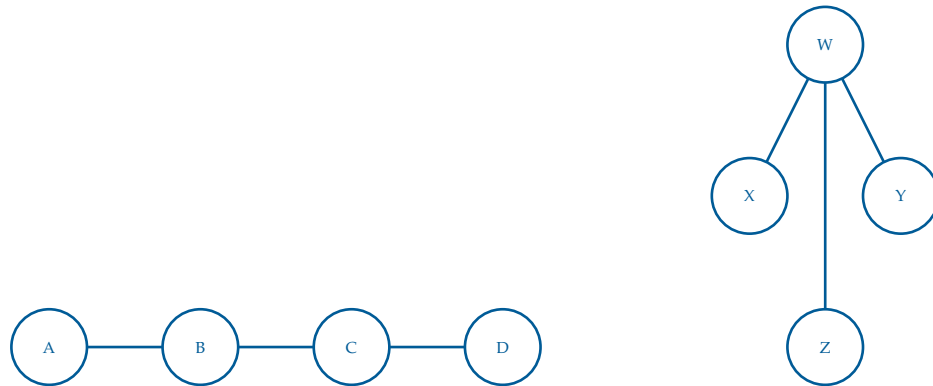


Figure 1.5: Graph X and Y; not isomorphic.

In Figure 1.5, we can see the vertices are not isomorphic without adding or subtracting edges. For example, Graph X does not have a vertex with 3 edges, unlike Graph Y.

If any of these differ, the graphs are not isomorphic:

- Number of vertices
- Number of edges
- Degree sequence (matching degree sequences do not guarantee isomorphism)
- Number of connected components
- Number of cycles of a given length
- Chromatic number

Example.

A common example that you will not be tested on but is important to know is the following. We may need to introduce a definition first.

Definition 5 (Clique). A clique is a set of vertices in a graph such that every pair of distinct vertices is connected by an edge.

Definition 6 (Clique Subgraph). A k -clique is exactly a subgraph isomorphic to the complete graph K_k . A 3-clique would be a clique of size 3, called K_3 .

Now, given two graphs G and H , how can we determine G contain a subgraph isomorphic to H , a graph of smaller to equal size to G ?

This may be easy if G and H are pretty small, but the math becomes much harder as the number of vertices and edges go up.

We can simplify this by transforming the inputs of a clique problem to inputs of the subgraph isomorphism problem.

The clique problem gives us a graph M and a threshold k . The output of this is whether M contains a clique of size at least k .

To transform this into an instance of the subgraph isomorphism problem, construct a smaller subgraph $H = K_k$, the complete graph on k vertices, and let $G = M$, the larger graph.

We now ask the subgraph isomorphism question:

Does G contain a subgraph isomorphic to H ?

Since H is the complete graph K_k , a subgraph of G isomorphic to H must contain k vertices where every pair of vertices is connected by an edge. By definition, this is exactly a k -clique.

Therefore,

$M \text{ contains a } k\text{-clique} \iff G \text{ contains a subgraph isomorphic to } K_k.$

This shows that every instance of the Clique problem can be transformed into an instance of the Subgraph Isomorphism problem. We call this an *algorithmic reduction*.²

1.5 Travelling Salesperson Problem and Nearest Neighbour

The Travelling Salesperson Problem (TSP) asks for the shortest route that visits every city exactly once and returns to the start. It is NP-hard: no fast algorithm is known that always finds the best route, and with just 20 cities there are about 6×10^{16} possible tours. A simple heuristic is the *nearest neighbour* method. Start at a city. Repeatedly go to the closest city you have not visited yet, and return to the start at the end. This is fast and often gives a reasonable route, but it may not be the shortest.

²Algorithmic Reductions are the basis of a set of problems called P, NP, and NP-Complete. See [this video](#) for more.

1.6 Topological Sort and DAGs

Another common use of graph algorithms is sorting them in a meaningful order. Specifically, we can sort directed graphs in a way that tells us exactly what order vertices need to be “visited” or operated on. We call this a *Topological Sorting* of the graph. A *topological order* is a linear ordering of the vertices in a graph such that for every directed edge $u \rightarrow v$ from vertex u to vertex v , vertex u comes before vertex v in the ordering. The most common usage for this is scheduling jobs, queueing tasks with dependencies, or creating tentative project schedules.

To do this, we are going to use Python’s Graphlib, and its TopologicalSorter module.

In the graphlib module, graphs are defined in the following format:

```
1 {
2   node: {predecessors}
3 }
```

A basic example of this is the following

```
1 {
2   "C": {"A", "B"}
3 }
```

Which says that A and B need to come before C, such that one possible ordering is

$$A \rightarrow B \rightarrow C$$

Now suppose we alter the graph format as follows:

```
1 {
2   "C": {"A", "B"},
3   "A": {"Z"},
4   "B": {"Z"},
5   "Z": {"C"}
6 }
```

Here,

- C depends on A and B
- A and B depend on Z

- Z depends on C

There is no way to determine a *starting point*. Now we have an ordering that contains a cycle. This implies that C must come before itself, which is impossible in a linear ordering. Therefore no topological ordering can exist.

Definition 7 (Cycle). A cycle is a valid walk $(v_1, v_2, \dots, v_n)$ where the only repeated vertex is $v_n = v_1$, meaning the last vertex equals the first vertex.

So, we cannot do a topological sort on a graph that contains a cycle. This provides us with a theorem:

Topological Sorting and DAGs

A directed graph admits a topological ordering if and only if it is a *Directed Acyclic Graph* (DAG). That is, the directed graph may not contain any cycles (acyclic).

In fact, the python graphlib library will not produce a valid Topological Sort if a cycle exists. Running the following code

```
1 from graphlib import TopologicalSorter
2 graph = {
3     "C": {"A", "B"},
4     "A": {"Z"},
5     "B": {"Z"},
6     "Z": {"C"}
7 }
8 list(TopologicalSorter(graph).static_order())
```

will produce the following error

```
1 graphlib.CycleError: ('nodes are in a cycle', ['C', 'Z', 'A', 'C'])
```

The exception reports one cycle that prevents the ordering from being constructed (the exact cycle printed may vary). In this case, $C \rightarrow Z \rightarrow A \rightarrow C$ forms a directed cycle.

We can also introduce tasks that can be done in parallel. Create the following file `parallel_dag.py`

```
1 from graphlib import TopologicalSorter
2 tasks = {
3     "Compile": set(),
4     "Build Frontend": {"Compile"},
5     "Build Backend": {"Compile"},
6     "Test": {"Build Frontend", "Build Backend"},
7     "Deploy": {"Test"},
8 }
```

```

8 }
9
10 ts = TopologicalSorter(tasks)
11 ts.prepare()
12
13 while ts.is_active():
14     ready = ts.get_ready()
15     print(ready)
16
17     for task in ready:
18         ts.done(task)
19

```

The 5 nodes here are tasks for a basic software project. The ordering is as follows:

Compile $\rightarrow$ Build Frontend and Build Backend $\rightarrow$ Test $\rightarrow$ Deploy

However notice that *Build Frontend* and *Build Backend* are not dependent on each other, so they could be done in *parallel*. The *Test* task cannot begin until both builds have finished. Running this code outputs the following

```

1 ('Compile',)
2 ('Build Frontend', 'Build Backend')
3 ('Test',)
4 ('Deploy',)

```

showing us that our sort outputs which ones can happen at the same time. In software development, many tasks have dependencies but can still be executed in parallel when those dependencies are satisfied. Topological sorting helps determine both the required execution order and which tasks may be performed concurrently.

1.7 Proofs

This section will be a set of proofs or algorithm design problems for graph-related concepts that are important to know. At the basis level, these are just practice problems for graph understand, but can be applied in many real world scenarios.

1.7.1 Example 1: Bipartite

Bipartite Graph

A *Bipartite Graph* is a graph $G = (V, E)$ such that V can be partitioned into two sets ($V = V_1 \cup V_2$) and ($V_1 \cap V_2 = \emptyset$) such that there are no edges between vertices in the same set.

Theorem 1. *An undirected graph G is bipartite if and only if it contains no cycles of odd length.*

Exercise 4 Algorithm for Determining Bipartite

Design pseudocode for an algorithm that determines whether or not an undirected graph is bipartite. *Hint: use the theorem.*

Working Space

Hint 2: This function may be useful

FindOdd(G, e)

Require: G , a graph on which DFS has been run; $\text{level}(v)$, the depth of each vertex v in the DFS tree; $e = (x, y)$, a back edge

Ensure: **true** if e creates a cycle of odd length, **false** otherwise

```

1: if  $\text{level}(x) - \text{level}(y) \equiv 0 \pmod{2}$ 
   then
2:   return true
3: else
4:   return false
5: end if
```

Answer on Page 118

While you do not have to go through it on your own, here is the outline of how we can do a proof by contradiction for a graph, specifically proving theorem 1.

Proof. First, suppose G is bipartite. Then its vertices can be partitioned into two sets V_1 and V_2 , with every edge going between V_1 and V_2 . Any cycle in G must alternate between vertices in V_1 and vertices in V_2 . Therefore, after an odd number of edges, the cycle would end in the opposite set from where it started, so it cannot return to its starting vertex. Hence every cycle has even length.

Conversely, suppose G contains no cycles of odd length. Choose any connected component of G , and pick a vertex s . Put each vertex at even distance from s in V_1 , and each vertex at odd distance from s in V_2 . We claim this gives a valid bipartition.

Assume for contradiction that there is an edge (u, v) where u and v are in the same set.

Then u and v have distances from s with the same parity. Taking shortest paths from s to u and from s to v , together with the edge (u, v) , gives a closed walk of odd length, and every odd closed walk contains an odd cycle. This contradicts the assumption that G has no odd cycles.

Thus no edge connects two vertices in the same set, so each connected component is bipartite. Applying this construction to every connected component of G , the whole graph G is bipartite.

Therefore, G is bipartite if and only if it contains no cycles of odd length. $\square$

1.7.2 Example 2 and 3. Cycle Detection and Leaves

Exercise 5 Cycle Detection

Design pseudocode for an algorithm which takes as an input an undirected graph, G , and an edge, $e = (u, v)$, and determines whether G has a cycle containing e .

Working Space

Answer on Page 118

Exercise 6 Leaf Removal

Prove the following statement: In any connected undirected graph, there is a vertex whose removal leaves the graph connected.

Working Space

Answer on Page 118

Sometimes, we may need to create new graphs as counterexamples to disprove a statement. Consider the following:

Definition 8 (Strongly Connected). *A directed graph G is strongly connected if every vertex is reachable from every other vertex.*

Strongly connected components are the largest subgraphs that are strongly connected. They come up when we study directed graphs, for example, when we want to know which groups of web pages can all reach each other.

Exercise 7 **Strongly Connected Graphs**

Create an example of a strongly connected directed graph $G = (V, E)$ such that, for every $v \in V$, removing v from G leaves a directed graph that is not strongly connected.

*Hint: the simplest answer is usually the best.
Try a single cycle.*

Working Space

Answer on Page 119

How the Internet Works

You tap a video, and it starts playing in less than a second, even though the server sending it may be 2,000 kilometers away. Meanwhile, your friend across the room is uploading a photo, and someone in Tokyo is joining a video call. The same wires and radio waves carry all of it, and no single computer is in charge.

So how does a message find its way from your phone to one particular computer out of billions? This chapter answers that question in four parts: messages travel as *packets*, every device has an *IP address*, routers pick paths using the *shortest-path algorithms* you already know, and *DNS* turns names like *youtube.com* into addresses. In the Web APIs chapter, you will use all of this to send HTTP requests.

2.1 A Network of Networks

The internet is not one network. It is tens of thousands of separate networks that have agreed to pass each other's messages along. Your home Wi-Fi is a tiny network. It connects to your internet service provider's network, which connects to bigger networks, which are joined by fiber-optic cables that run across continents and along the ocean floor.

The devices that join networks together are called *routers*. Your home has one: the box with the blinking lights. Each router has a simple job. When a message arrives, it looks at where the message is going and sends it one step closer. No router knows the whole journey.

2.2 Packets

The old telephone network worked by reserving a wire: while you were on a call, a complete circuit from your phone to your friend's phone belonged to you, even during silences. The internet works differently. Every message is chopped into small pieces called *packets*, usually no more than about 1,500 bytes each. Each packet travels on its own, sharing the cables with everyone else's packets.

A packet is like a postcard. Its *header* is the address side: where it came from, where it is going, and a sequence number saying which piece of the message it is. Its *payload* is the message side.

Example: You send a 3 MB photo (3,000,000 bytes). If each packet carries 1,460 bytes of

payload, how many packets is that?

Solution: $3,000,000/1,460 \approx 2,054.8$, so it takes 2,055 packets. (The last one is not full.)

Sharing is what makes the internet efficient, but it has a cost. Packets from the same message can take different routes, arrive out of order, or get lost when a busy router runs out of room and drops them.

2.2.1 TCP: Getting It All, in Order

Most of the time, you need every byte: a missing piece of a web page or a photo is a disaster. The *Transmission Control Protocol* (TCP) handles this. The receiver uses the sequence numbers to put the packets back in order and to notice which ones never arrived. It sends back *acknowledgments* of what it received, and the sender sends the missing packets again.

Let us simulate it. Create a file called `packets.py`:

```
1 import random
2
3 random.seed(4)
4 message = "Meet me at the library at 3:30, after band practice."
5 size = 8 # characters per packet
6
7 # Sender: chop the message into numbered packets
8 packets = []
9 for start in range(0, len(message), size):
10     packets.append((start // size, message[start:start + size]))
11 print("Sending", len(packets), "packets")
12
13 # Receiver: keep the packets that arrive, keyed by number
14 received = {}
15 needed = list(range(len(packets)))
16 round_number = 1
17
18 while needed:
19     in_flight = [packets[n] for n in needed]
20     random.shuffle(in_flight) # packets take different routes
21     arrived = []
22     for number, data in in_flight:
23         if random.random() < 0.25: # 1 packet in 4 gets lost
24             continue
25         received[number] = data
26         arrived.append(number)
27     needed = [n for n in range(len(packets)) if n not in received]
28     print("Round", round_number, "arrived:", arrived, "still missing:", needed)
29     round_number += 1
30
```

```

31 # Put the packets back in order
32 print("".join(received[n] for n in range(len(packets))))

```

When you run it, you should see:

```

Sending 7 packets
Round 1 arrived: [6, 3, 0, 1] still missing: [2, 4, 5]
Round 2 arrived: [2, 5, 4] still missing: []
Meet me at the library at 3:30, after band practice.

```

The packets arrived scrambled, and three of them were lost, yet the message came out perfectly. (Real networks lose far fewer than 1 packet in 4; we exaggerated so you can see it happen.)

2.2.2 UDP: Fast, No Promises

Resending takes time. For a video call or an online game, a packet that arrives half a second late is worthless: the moment has passed. These programs use the *User Datagram Protocol* (UDP), which just sends packets and hopes. If one is lost, the video glitches for an instant and moves on.

2.3 IP Addresses

Every device on the internet has an *IP address*, and every packet's header carries the source and destination addresses. (IP stands for Internet Protocol.) The classic kind, *IPv4*, is written as four numbers separated by dots:

192.168.1.23

Each of the four numbers fits in one byte, so it is between 0 and 255, and the whole address is 32 bits. That allows $2^{32} = 4,294,967,296$ addresses, about 4.3 billion.

That sounded like plenty in 1981. Today, there are far more phones, laptops, game consoles, and smart doorbells than that. Two fixes keep things running:

- **Private addresses.** Addresses that start with 192.168 or 10 are reserved for use inside a home or school network. Your home router has one public address on the internet and hands out private addresses to every device inside. When your laptop sends a packet out, the router swaps in its public address and remembers where to send the reply. This trick is called *network address translation* (NAT).

- **IPv6.** The new kind of address is 128 bits long, written in hexadecimal, like this one:

2001:db8::8a2e:370:7334

That allows $2^{128} \approx 3.4 \times 10^{38}$ addresses, enough to never run out.

2.3.1 Networks Have Prefixes

Addresses are not handed out at random. A whole network gets a block of addresses that share the same beginning, called a *prefix*. The block written `192.168.1.0/24` means “every address whose first 24 bits match `192.168.1`”. Since 24 bits is three bytes, that is every address from `192.168.1.0` to `192.168.1.255`: 256 addresses.

A prefix works like a ZIP code. A mail sorter in Chicago does not need to know every street in Denver; it only needs to know that everything starting with `802` goes to Denver. In the same way, a router far away needs only one entry for a whole network, not one for every device in it.

Python’s built-in `ipaddress` module understands all of this. Try it in the interpreter:

```
1 >>> import ipaddress
2 >>> ip = ipaddress.ip_address("192.168.1.23")
3 >>> net = ipaddress.ip_network("192.168.1.0/24")
4 >>> ip in net
5 True
6 >>> net.num_addresses
7 256
8 >>> int(ip)
9 3232235799
```

The last line shows what the address really is to a computer: one 32-bit number. The dotted form is just easier for people to read.

2.3.2 Ports

An address gets a packet to the right computer, but a computer runs many programs at once. A *port* number, also in the header, says which program the packet is for. Web servers listen on port 443 (or 80 for unencrypted pages). In the Building a Web API chapter, your server will listen on port 3000, which is why you will visit `localhost:3000`. The name `localhost` always means “this computer”, at the special address `127.0.0.1`.

2.4 Routing: Shortest Paths for Packets

Now the central question: how does a router decide which way to send a packet?

Picture the routers as the nodes of a graph and the cables between them as the edges. Give each edge a cost, such as the time in milliseconds a packet takes to cross it. Finding a good route is then exactly the problem from the Finding Shortest Paths chapter.

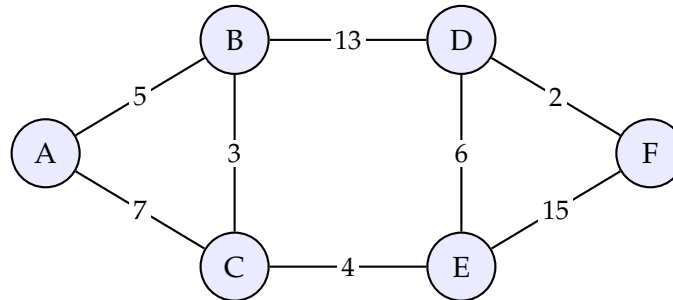


Figure 2.1: A small network of six routers. Each number is the delay of that link in milliseconds.

A router does not need the whole path. It only needs to know, for each destination, which neighbor to hand the packet to: the *next hop*. The list of next hops is the router's *forwarding table*. To build it, the router runs Dijkstra's algorithm from itself, then follows the prev links back from each destination until it reaches the first step.

Example: Find router A's next hop to F.

Solution: Running Dijkstra from A gives these costs: B 5, C 7, E 11 (through C), D 17 (through C and E), and F 19 (through D). The best path is A → C → E → D → F. Router A only needs the first step: send to C. Router C will make its own decision when the packet arrives.

Here is the same computation in Python. Create `routing.py`:

```

1 import math
2
3 # Each router lists its neighbors and the delay (in ms) of each link
4 links = {
5     "A": {"B": 5, "C": 7},
6     "B": {"A": 5, "C": 3, "D": 13},
7     "C": {"A": 7, "B": 3, "E": 4},
8     "D": {"B": 13, "E": 6, "F": 2},
9     "E": {"C": 4, "D": 6, "F": 15},
10    "F": {"D": 2, "E": 15},
11 }
12
13 def dijkstra(links, origin):
14     dist = {router: math.inf for router in links}

```

```

15     dist[origin] = 0
16     prev = {}
17     unvisited = set(links)
18     while unvisited:
19         current = min(unvisited, key=lambda r: dist[r])
20         if dist[current] == math.inf:
21             break
22         unvisited.remove(current)
23         for neighbor, delay in links[current].items():
24             if dist[current] + delay < dist[neighbor]:
25                 dist[neighbor] = dist[current] + delay
26                 prev[neighbor] = current
27     return dist, prev
28
29 def forwarding_table(links, router):
30     dist, prev = dijkstra(links, router)
31     table = {}
32     for destination in sorted(links):
33         if destination == router or destination not in prev:
34             continue
35         # Walk back from the destination until the step just after the router
36         hop = destination
37         while prev[hop] != router:
38             hop = prev[hop]
39         table[destination] = (hop, dist[destination])
40     return table
41
42 def show(table):
43     for destination, (hop, cost) in table.items():
44         print(f"  to {destination}: send to {hop}  ({cost} ms)")
45
46 print("Router A's forwarding table:")
47 show(forwarding_table(links, "A"))
48
49 # The link between C and E breaks
50 del links["C"]["E"]
51 del links["E"]["C"]
52 print("After the C-E link fails:")
53 show(forwarding_table(links, "A"))

```

The `dijkstra` function is the one from the Finding Shortest Paths chapter, written more compactly: `min(unvisited, key=...)` finds the unvisited router with the lowest cost in one line.

When you run it, you should see:

```

Router A's forwarding table:
  to B: send to B  (5 ms)
  to C: send to C  (7 ms)
  to D: send to C  (17 ms)
  to E: send to C  (11 ms)

```

```
to F: send to C (19 ms)
After the C-E link fails:
to B: send to B (5 ms)
to C: send to C (7 ms)
to D: send to B (18 ms)
to E: send to B (24 ms)
to F: send to B (20 ms)
```

When the C–E cable breaks (perhaps a backhoe digs it up), the routers notice, recompute, and traffic flows around the damage through B and D. This ability to route around failures is why the internet has no single point that can bring the whole thing down.

2.4.1 Two Ways to Share the Map

For Dijkstra to work, each router needs the map. Real networks use both of the algorithms from the Finding Shortest Paths chapter:

- In *link-state routing*, every router tells every other router about its own links. Each router then has the whole map and runs Dijkstra’s algorithm itself. The protocol called OSPF works this way, inside large company and university networks.
- In *distance-vector routing*, each router only talks to its neighbors, telling them “here is how far I am from every destination.” Each router updates its own distances from what its neighbors report, round after round. That is the Bellman-Ford algorithm, spread across many machines. The protocol called RIP works this way.

Between the giant networks that make up the internet, the rules are different. Those networks belong to competing companies, and their routers (speaking a protocol called BGP) choose paths based on business agreements as well as speed. The shortest path is not always the one taken.

2.4.2 The Speed of Light Is a Speed Limit

Light in an optical fiber travels at about 200,000 kilometers per second, two-thirds of its speed in a vacuum. That sets a hard limit on how fast any message can go.

Example: New York and London are about 5,570 km apart. What is the shortest possible time for a packet to go there and back?

Solution: One way takes $5,570/200,000 = 0.0279$ seconds, about 28 ms. A round trip takes at least 56 ms. Real cables do not run in perfectly straight lines and routers add delays, so the real time is more like 70 ms. No amount of engineering can beat 56 ms: that is physics.

This is why online games put servers in many cities, and why video sites keep copies of popular videos close to you.

2.5 DNS: The Internet's Phone Book

Nobody wants to type 142.250.72.14 to watch a video. People remember names like youtube.com instead. The *Domain Name System* (DNS) translates names into IP addresses.

DNS is not one giant list; it is a hierarchy, read from right to left:

1. Your computer asks a *resolver*, usually run by your internet provider, for www.example.com.
2. If the resolver does not know, it asks a *root server*: “Who handles .com?” There are 13 named root servers, each copied to many places around the world.
3. The root server points to the .com servers, which point to the servers for example.com.
4. The example.com servers give the final answer: the IP address of www.example.com.

That is a lot of asking for one web page. To make it fast, every answer comes with a *time to live* (TTL), and resolvers *cache* answers until the TTL runs out. Popular names are almost always already in the cache, so most lookups take a few milliseconds.

You can look up names yourself. In a terminal, `nslookup example.com` prints the address. In Python:

```
1 >>> import socket
2 >>> socket.gethostbyname("localhost")
3 '127.0.0.1'
4 >>> socket.gethostbyname("example.com")
```

The second lookup needs an internet connection, and the address it prints can change over time, because the owners of a name can point it at a different server whenever they like. That is one of the best things about DNS: a website can move to a new computer, and nobody has to learn a new address.

2.6 Putting It Together

Here is what happens when you type `https://www.example.com/` into your browser and press Enter:

1. **DNS:** The browser looks up www.example.com and gets an IP address.

2. **TCP:** Your computer and the server exchange three small packets to open a connection, agreeing on starting sequence numbers.
3. **Encryption:** Because the address starts with https, the two sides agree on a secret key, using public-key cryptography, the family of ideas behind RSA from the Number Theory and Cryptography chapter. From now on, routers along the way can see where your packets are going but not what is in them. That is what the lock icon means.
4. **HTTP:** The browser sends a request, “GET this page, please.” You will learn the details of HTTP in the Web APIs chapter.
5. **Routing:** Every packet, in both directions, is passed from router to router, each one choosing a next hop from its forwarding table.
6. **Reassembly:** TCP puts the reply’s packets back in order, asks again for any that were lost, and hands the complete page to the browser, which draws it.

All of this usually takes less than a second.

Notice how the jobs are stacked. HTTP does not care how packets are routed, and routers do not care what is in the packets. Engineers call each job a *layer*. Each layer wraps the one above it, like a letter in an envelope inside a bigger envelope:

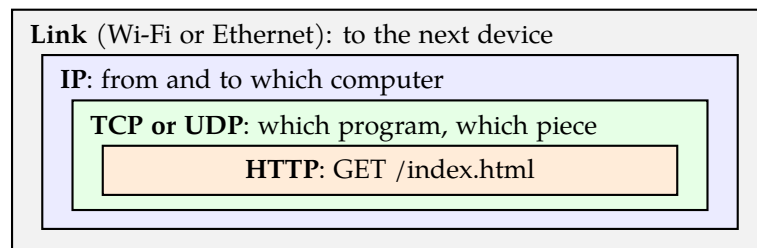


Figure 2.2: The layers of a packet. Each layer adds its own header around the layer inside it.

Exercise 8 Counting Packets

A song file is 5 MB (5,000,000 bytes). If each packet carries 1,460 bytes of payload, how many packets does it take to send it?

Working Space

Answer on Page 119

Exercise 9 Video Calls

Why does a video call usually use UDP instead of TCP?

Working Space

Answer on Page 119

Exercise 10 Sizes of Networks

How many addresses are in the block 10.0.0.0/16? How many are in a /8 block?

Working Space

Answer on Page 119

Exercise 11 Enough for Everyone?

There are about 8.1 billion people on Earth. If all 2^{128} IPv6 addresses were shared equally, how many would each person get?

Working Space

Answer on Page 120

Exercise 12 Router D's Table

Using the network in Figure 2.1, find router D's next hop and cost to each other router. Check your answer by

Working Space

changing the last lines of `routing.py`.

Answer on Page 120

Exercise 13 **Around the World**

Denver and Tokyo are about 9,330 km apart. What is the shortest possible round-trip time for a packet between them through optical fiber? Why might an online game feel laggy for a player in Denver on a server in Tokyo?

Working Space

Answer on Page 120

Exercise 14 **Caching Trade-off**

A website sets the TTL of its DNS records to one week. What is good about that? What could go wrong if the site moves to a new server with a new IP address?

Working Space

Answer on Page 120

Exercise 15 **Trace a Route**

In a terminal, run `tracert example.com` (on Windows, `tracert example.com`). Each line is one router along the way, with the time it took to reply. How many hops does it take? Do you see a place where the time jumps suddenly?

Working Space

What might cause that?

Answer on Page 120

HTML

Right-click any web page and choose “View Page Source.” The wall of angle brackets you see is HTML, and every site you have ever visited, from your school’s homepage to YouTube, is built on it. In this chapter, you will write a recipe page from scratch, one feature at a time.

HTML, an abbreviation for Hypertext Markup Language, is the standard language for creating web pages and web applications. It is a cornerstone technology of the World Wide Web and forms the structure and layout of web content.

3.1 HTML Elements

An HTML document is composed of a series of elements, which are denoted by tags. Elements have an opening tag and a closing tag with content in between. Some elements, however, are self-closing and do not contain any content.

An element consists of three parts:

- **Opening tag:** The opening tag is the start of an element. The format of it is `<tagname>`.
- **Content:** Content is the data or information between the tags.
- **Closing tag:** The closing tag indicates the end of an element. The format of it is `</tagname>`.

For example, the paragraph tag `<p>` is used to denote a paragraph:

```
1 <p>This is a paragraph.</p>
```

This applies to a lot of HTML elements but not every one. There are HTML elements that are self-closing. A self-closing element does not require a closing tag. The image tag `<img>` is an example of a self-closing element.

```
1 
```

3.2 Attributes

Attributes provide additional information about HTML elements. For some elements like `img`, they are required for the element to function correctly. They are always specified in the opening tag and consist of a name and value pair.

The syntax for attributes is:

```
1 <tagname attribute="value">Content</tagname>
```

There are so many attributes available, we cannot cover them all here. However, some common attributes that work on most elements include:

- **id**: The `id` attribute specifies a unique identifier for an element (should be unique within the entire page)
- **class**: The `class` attribute specifies one or more class names for styling with CSS
- **style**: The `style` attribute specifies inline CSS styling
- **title**: The `title` attribute specifies additional information displayed as a tooltip

Examples:

```
1 <p id="intro" class="highlight">
2   Introduction paragraph
3 </p>
4
5 <div style="color: blue; font-size: 16px;">
6   Styled text
7 </div>
8
9 
```

3.3 HTML Document Structure

A typical HTML document has a specific structure. Every HTML5 document begins with a DOCTYPE declaration, which tells the browser which version of HTML to expect.

The basic structure of an HTML document is:

```
1 <!DOCTYPE html>
2 <html>
```

```
3 <head>
4   <title>Page Title</title>
5 </head>
6 <body>
7   <!-- Content goes here -->
8 </body>
9 </html>
```

The main components are:

- **DOCTYPE declaration:** This informs the browser about the version of HTML. For HTML5, it is `<!DOCTYPE html>`.
- **html:** The `html` element is the root element that wraps all other content.
- **head:** This element contains metadata and information about the page, including:
 - `<title>`: This element sets the title of the page, which appears in the browser tab.
 - `<meta>`: This element includes metadata such as character encoding, and viewport settings.
- **body:** The `body` element contains all the visible content of the web page.

Here is a more complete example:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>My Website</title>
7 </head>
8 <body>
9   <h1>Welcome</h1>
10  <p>This is my website.</p>
11 </body>
12 </html>
```

Exercise 16 Building a Full HTML Document

Create the beginning of a web page for a chocolate chip cookie recipe. Add the DOCTYPE declaration, the `html` element, a head with a title that says "Chocolate Chip Cookies", and move the following content into the body.

Working Space

```
1 <body>
2   <h1 id="recipe-title">Chocolate Chip
   ↳ Cookies</h1>
3   <p class="intro" title="A short recipe
   ↳ description">
4     This recipe makes soft and sweet
   ↳ chocolate chip cookies.
5   </p>
6 </body>
```

Write a full HTML5 page that wraps this content correctly.

Answer on Page 120

After this exercise, the page should look like the following:

Chocolate Chip Cookies

This recipe makes soft and sweet chocolate chip cookies.

Figure 3.1: HTML Structure Exercise Output

3.4 Web Development and HTML

The Web Development trio consists of three essential languages that work together to create modern web applications. These languages are HTML, CSS, and JavaScript.

- **HTML:** HTML provides the structure and content of web pages. It defines what elements exist on the page. Think of it as the skeleton and meat of a web page.
- **CSS:** Cascading Style Sheets (CSS) handles the presentation and styling. It controls

how elements look, including colors, fonts, spacing, and layout. Think of it as the clothing and makeup that makes the page visually appealing.

- **JavaScript:** JavaScript adds interactivity and behavior. It enables dynamic content changes, form validation, animations, and user interactions. Think of it as the brain that allows the page to respond to actions like clicking a button or submitting a form.

This chapter is about HTML. In the CSS and JavaScript chapter, you will take the recipe page you build here, style it, and make it respond to clicks.

3.5 Metadata

Metadata is information about the HTML document that is not directly displayed on the page. The information is provided through a series of `<meta>` tags with certain attributes and their corresponding values. These are placed in the `<head>` section and provides important information to browsers, search engines, and developers.

Common metadata elements include:

- `<meta charset="UTF-8">`: This line specifies the character encoding of the document. UTF-8 is the standard encoding for web pages.
- `<meta name="viewport" content="width=device-width, initial-scale=1.0">`: This line configures the viewport for responsive design. This tells mobile browsers how to scale the page.
- `<meta name="description" content="...">`: This line provides a brief description of the page content for search engines.
- `<meta name="keywords" content="...">`: This line lists keywords for the page. (Major search engines now ignore it, because sites abused it.)
- `<meta name="author" content="...">`: This line specifies the author of the document.

Example of using multiple meta tags:

```
1 <head>
2   <meta charset="UTF-8">
3   <meta name="viewport" content="width=device-width, initial-scale=1.0">
4   <meta name="description" content="A tutorial about web development">
5   <meta name="keywords" content="HTML, CSS, JavaScript">
6   <meta name="author" content="John Doe">
7   <title>Web Development Tutorial</title>
8 </head>
```

3.6 Div Elements

The `<div>` element is a generic container used to group content together. The name “div” stands for “division.” `<div>` has no specific meaning to it. It is a neutral wrapper used for layout, styling, or organizing content with CSS and JavaScript.

The `<div>` element is one of the most commonly used HTML elements in existing applications. It is often used with `class` or `id` attributes to apply styling or add behavior.

Here is a basic example:

```
1 <div class="container">
2   <h2>Section Title</h2>
3   <p>This is some content inside a div.</p>
4 </div>
```

As mentioned, it is one of the most commonly used HTML elements. Here is an example with multiple divs for layout:

```
1 <div class="header">
2   <h1>Website Title</h1>
3 </div>
4
5 <div class="content">
6   <p>Main content goes here.</p>
7 </div>
8
9 <div class="footer">
10  <p>&copy; 2024 My Website</p>
11 </div>
```

Exercise 17 Grouping Recipe Content with div

Group the recipe title and introduction paragraph into a div. Add a “class” attribute to the div with the value “recipe-header”.

Working Space

```
1 <body>
2   <h1 id="recipe-title">Chocolate Chip
3     ↳ Cookies</h1>
4   <p class="intro" title="A short recipe
5     ↳ description">
6     This recipe makes soft and sweet
7     ↳ chocolate chip cookies.
```

```
5 </p>  
6 </body>
```

Add two more `p` elements for ingredients and steps in the body. Place the `p` elements into two separate `div` elements. The ingredients `div` should have a class attribute with a value of "ingredients-section". The steps `div` should have a class attribute with a value of "steps-section".

Answer on Page 121

After this exercise, the page should now look like the following:

Chocolate Chip Cookies

This recipe makes soft and sweet chocolate chip cookies.

Ingredients

Steps

Figure 3.2: HTML Div Exercise Output

3.7 Semantic HTML

Unlike `<div>` elements, Semantic HTML elements are elements that clearly describe their meaning to both the browser and the developer. This improves code readability, helps with search engine optimization (SEO), and makes pages more accessible to assistive technologies like screen readers.

Semantic elements follow a logical structure that reflects the content of the page. Some common semantic elements include:

- `<header>`: The header element represents the top section of the page or a section, typically containing the logo, navigation, or introductory content.
- `<nav>`: The nav element contains navigation links to other pages or sections.
- `<main>`: The main element specifies the main content area of the page.
- `<article>`: The article element represents independent, self-contained content that could stand alone, such as a blog post or news article.
- `<section>`: The section element defines a generic section of content. This is preferred instead of using multiple divs for grouping related content.
- `<aside>`: The aside element contains supplementary content like sidebars, ads, or related links.
- `<footer>`: The footer element represents the bottom section of a page or section.

Here is an example of using semantic elements to structure a simple blog page:

```
1 <header>
2   <h1>My Blog</h1>
3   <nav>
4     <a href="/">Home</a>
5     <a href="/about">About</a>
6     <a href="/contact">Contact</a>
7   </nav>
8 </header>
9
10 <main>
11   <article>
12     <h2>Article Title</h2>
13     <p>Article content here...</p>
14   </article>
15
16   <aside>
17     <h3>Related Posts</h3>
18     <ul>
19       <li><a href="#post1">Post 1</a></li>
20       <li><a href="#post2">Post 2</a></li>
21     </ul>
22   </aside>
23 </main>
24
25 <footer>
26   <p>&copy; 2026 My Blog. All rights reserved.</p>
27 </footer>
```

Exercise 18 Using Semantic Elements

Replace general `div` containers with semantic elements where they make sense. Use `header` for the “recipe-header”. Use `section` for the “ingredients-section” and “steps-section”. Afterwards, put these inside a `main` element. Add a `footer` for the bottom note. There will be no content in the footer for now.

Working Space

```
1 <body>
2   <div class="recipe-header">
3     <h1 id="recipe-title">Chocolate Chip
      ↳ Cookies</h1>
4     <p class="intro" title="A short recipe
      ↳ description">
5       This recipe makes soft and sweet
        ↳ chocolate chip cookies.
6     </p>
7   </div>
8
9   <div class="ingredients-section">
10    <p>Ingredients</p>
11  </div>
12
13  <div class="steps-section">
14    <p>Steps</p>
15  </div>
16 </body>
```

Answer on Page 121

After this, the web page should still look the same from the previous exercise.

3.8 Headings and Sections

HTML provides six levels of headings, ranging from `<h1>` to `<h6>`. Headings establish a hierarchical structure for page content. `<h1>` is the most important and typically used once per page for the main title, while `<h6>` is the least important.

```
1 <h1>Main Page Title</h1>
2 <h2>Major Section</h2>
3 <h3>Subsection</h3>
4 <h4>Sub-subsection</h4>
5 <h5>Minor heading</h5>
6 <h6>Least important heading</h6>
```

Headings are important for defining the structure of the content and improving accessibility. They allow users to quickly scan the page and understand the organization of information. Proper use of headings also benefits SEO by signaling the importance of content to search engines.

Exercise 19 Adding Section Headings

Replace the p elements in the ingredients and steps sections with h2 tags.

Working Space

```
1 <main>
2   <section class="ingredients-section">
3     <p>Ingredients</p>
4   </section>
5
6   <section class="steps-section">
7     <p>Steps</p>
8   </section>
9 </main>
```

Update the main content to include headings both the Ingredients and Steps sections. Add an id attribute to each heading with the value "ingredients" and "steps" respectively.

Answer on Page 122

After this exercise, the page should now look like the following:

Chocolate Chip Cookies

This recipe makes soft and sweet chocolate chip cookies.

Ingredients

Steps

Figure 3.3: HTML Headings Exercise Output

3.9 Text Elements

HTML provides several elements for formatting text content. Here is a list of some of the most commonly used ones:

- `<p>`: The `p` element defines a paragraph. Browsers automatically add space before and after.
- `<br>`: The `br` element inserts a line break. This is a self-closing element.
- `<hr>`: The `hr` element displays a horizontal line. This is a self-closing element.
- `<strong>`: The `strong` element indicates strong importance. This is typically displayed in bold.
- `<em>`: The `em` element indicates emphasis. This is typically displayed in italics.
- `<b>`: The `b` element displays bold text without semantic importance.
- `<i>`: The `i` element displays italic text without semantic emphasis.
- `<mark>`: The `mark` element highlights text with a yellow background.
- `<code>`: The `code` element displays text in monospace font for code.

Examples:

```
1 <p>This is a <strong>important</strong> point.</p>
2
3 <p>This word is <em>emphasized</em>.</p>
4
```

```
5 <p><code>var x = 5;</code></p>
6
7 <p>This is <mark>highlighted</mark> text.</p>
```

Exercise 20 Using Text Elements

Working Space

Add a `p` element in the footer section of the cookie recipe page. The paragraph should say: **"Important Note: Make sure not to overbake the cookies, or they will become hard."**

```
1 <footer>
2 </footer>
```

Add a paragraph element inside of the footer element. Use the `strong` element to make "Important Note:" bold.

Answer on Page 122

After this exercise, the page should now look like the following:

Chocolate Chip Cookies

This recipe makes soft and sweet chocolate chip cookies.

Ingredients

Steps

Important Note: Make sure not to overbake the cookies, or they will become hard.

Figure 3.4: HTML Text Exercise Output

3.10 Links

Links are created using the `<a>` (anchor) element with the `href` attribute specifying the destination.

```
1 <a href="https://www.example.com">Visit Example</a>
```

Links can point to various destinations:

- **External websites:** `<a href="https://www.google.com">External Site</a>`. This will navigate the user to Google.
- **Other pages within the same site:** `<a href="about.html">About</a>`. This will navigate the user to the `about.html` page within the same website.
- **Email addresses:** `<a href="mailto:email@example.com">Email</a>`. This will open the user's email to send an email to the specified address.
- **Phone numbers:** `<a href="tel:+1234567890">Call Us</a>`. This will initiate a phone call on devices that support it. This can be useful for mobile users.
- **Specific locations on the same page:** `<a href="#section2">Jump to Section 2</a>` (where `<h2 id="section2">Section 2</h2>`). When the link is clicked, the page will scroll to the header with the id `"section2"`.

Apart from the `href` attribute, here are some other useful attributes for links:

- `target="_blank"`: This attribute and value pair opens the link in a new tab.
- `target="_self"`: This attribute and value pair opens the link in the same tab. This is the normal behavior if the `target` attribute is not specified.
- `title="tooltip"`: This attribute and value pair displays text when the user hovers over the link like a tooltip.

3.11 Lists

HTML supports two main types of lists to organize content. These include unordered lists and ordered lists. Both types of lists use the `<li>` element to define individual list items.

3.11.1 Unordered Lists

Unordered lists display items with bullet points. As the name suggests, the order of items is not important like the items of a shopping list. They use the `<ul>` tag with list items defined by `<li>`:

```
1 <ul>
2   <li>Apples</li>
3   <li>Bananas</li>
4   <li>Oranges</li>
5 </ul>
```

This would render an unordered list as:

- Apples
- Bananas
- Oranges

3.11.2 Ordered Lists

Ordered lists display items with numbers or other counters. As the name suggests, the order of items is important, like steps in a recipe. They use the `<ol>` tag:

```
1 <ol>
2   <li>First step</li>
3   <li>Second step</li>
4   <li>Third step</li>
5 </ol>
```

The output of the above code would be:

1. First step
2. Second step
3. Third step

The `<ol>` tag supports the `type` attribute to customize or change the default numbering style. So instead of numbers, you can use letters or Roman numerals. Some of the common attributes include:

- type="a": Lowercase letters (a, b, c...).
- type="A": Uppercase letters (A, B, C...).
- type="i": Lowercase Roman numerals (i, ii, iii...)
- type="I": Uppercase Roman numerals (I, II, III...)

Exercise 21 Ingredients and Steps Lists

Use lists to show the cookie ingredients and the baking steps. The ingredients should be an unordered list, and the steps should be an ordered list.

Working Space

```

1 <section class="ingredients-section">
2   <h2 id="ingredients">Ingredients</h2>
3   <!-- Add an unordered list of ingredients
4     ↳ here -->
5 </section>
6
7 <section class="steps-section">
8   <h2 id="steps">Steps</h2>
9   <!-- Add an ordered list of steps here
    ↳ -->
10 </section>

```

Here is the list of ingredients:

- 2 1/4 cups flour.
- 1 teaspoon baking soda.
- 1 teaspoon salt.
- 1 cup butter, 3/4 cup sugar.
- 3/4 cup brown sugar, 2 eggs.
- 2 cups chocolate chips.

Here is the list of steps:

1. Preheat the oven to 375°F.
2. Mix the dry ingredients in one bowl.
3. Mix the butter, sugar, brown sugar,

and eggs in another bowl.

4. Stir the dry ingredients into the wet ingredients.
5. Add the chocolate chips.
6. Scoop the dough onto a baking sheet.
7. Bake for 9 to 11 minutes.

Answer on Page 122

After this exercise, the page should now look like the following:

Chocolate Chip Cookies

This recipe makes soft and sweet chocolate chip cookies.

Ingredients

- 2 1/4 cups flour
- 1 teaspoon baking soda
- 1 teaspoon salt
- 1 cup butter
- 3/4 cup sugar
- 3/4 cup brown sugar
- 2 eggs
- 2 cups chocolate chips

Steps

1. Preheat the oven to 375°F.
2. Mix the dry ingredients in one bowl.
3. Mix the butter, sugar, brown sugar, and eggs in another bowl.
4. Stir the dry ingredients into the wet ingredients.
5. Add the chocolate chips.
6. Scoop the dough onto a baking sheet.
7. Bake for 9 to 11 minutes.

Important Note: Make sure not to overbake the cookies, or they will become hard.

Figure 3.5: HTML Lists Exercise Output

3.12 Navigation

The `<nav>` element is used to wrap navigational links within a navigation section. It is a semantic element that helps structure the page and signal to assistive technologies that this section contains navigation. This is commonly used for site menus, table of contents, or any set of links that help users navigate the website.

```

1 <nav>
2   <ul>
3     <li><a href="/">Home</a></li>
4     <li><a href="/about">About</a></li>
5     <li><a href="/services">Services</a></li>
6     <li><a href="/contact">Contact</a></li>
7   </ul>
8 </nav>

```

Exercise 22 Creating a Simple Navigation Bar

Add a navigation bar to the top of the cookie page within the header element. The navigation should link to the “Ingredients” and “Steps” sections.

Working Space

```

1 <header>
2   <h1 id="recipe-title">Chocolate Chip
3     ↳ Cookies</h1>
4   <p class="intro" title="A short recipe
5     ↳ description">
6     This recipe makes soft and sweet
7     ↳ chocolate chip cookies.
8   </p>
9
10  <!-- Add navigation here -->
11 </header>

```

Insert a `nav` element with links to each section. Each link should use the `href` attribute to point to the corresponding section’s id (e.g., `href="#ingredients"` and `href="#steps"`) within the same page.

Answer on Page 123

After this exercise, the page should now look like the following:

Chocolate Chip Cookies

This recipe makes soft and sweet chocolate chip cookies.

- [Ingredients](#)
- [Steps](#)

Ingredients

- 2 1/4 cups flour
- 1 teaspoon baking soda
- 1 teaspoon salt
- 1 cup butter
- 3/4 cup sugar
- 3/4 cup brown sugar
- 2 eggs
- 2 cups chocolate chips

Steps

1. Preheat the oven to 375°F.
2. Mix the dry ingredients in one bowl.
3. Mix the butter, sugar, brown sugar, and eggs in another bowl.
4. Stir the dry ingredients into the wet ingredients.
5. Add the chocolate chips.
6. Scoop the dough onto a baking sheet.
7. Bake for 9 to 11 minutes.

Important Note: Make sure not to overbake the cookies, or they will become hard.

Figure 3.6: HTML Navigation Exercise Output

3.13 Tables

Tables organize data into rows and columns. They are created with the `<table>` tag, which acts as a container for the table parts that make up the layout. Those table elements include:

- `<tr>`: The `tr` element defines a table row
- `<th>`: The `th` element defines a table header cell. By default, it is displayed in bold and centered. It is used to indicate that the cell is a header for a column or row.
- `<td>`: The `td` element defines a table data cell. It is used for regular cells that contain data.

- `<thead>`: The `thead` element groups header rows.
- `<tbody>`: The `tbody` element groups body rows.
- `<tfoot>`: The `tfoot` element groups footer rows.
- `<caption>`: The `caption` element provides a title for the table.

Here is a basic table example:

```

1  <table>
2    <caption>Monthly Sales</caption>
3    <thead>
4      <tr>
5        <th>Month</th>
6        <th>Sales</th>
7      </tr>
8    </thead>
9    <tbody>
10     <tr>
11       <td>January</td>
12       <td>$5000</td>
13     </tr>
14     <tr>
15       <td>February</td>
16       <td>$6000</td>
17     </tr>
18   </tbody>
19   <tfoot>
20     <tr>
21       <th>Total</th>
22       <td>$11000</td>
23     </tr>
24   </tfoot>
25 </table>

```

The output of the above code would be:

Table 3.1: Monthly Sales

Month	Sales
January	\$5000
February	\$6000
Total	\$11000

There are attributes that can be applied to table cells to make them span multiple rows or columns. These attributes are `rowspan` (row span) and `colspan` (column span). These attributes are used to merge cells together, which can be useful for creating more complex table layouts. `colspan` is used to make a cell span across multiple columns, while `rowspan`

is used to make a cell span across multiple rows. The value of these attributes specifies the number of rows or columns the cell should span.

Here is an example of how to use them:

```

1 <table>
2   <tr>
3     <th colspan="2">Personal Information</th>
4   </tr>
5   <tr>
6     <td>Name</td>
7     <td>John Doe</td>
8   </tr>
9   <tr>
10    <th>Age</th>
11    <th rowspan="2">Skills</th>
12  </tr>
13  <tr>
14    <td>30</td>
15  </tr>
16  <tr>
17    <td colspan="2">HTML, CSS, JavaScript</td>
18  </tr>
19 </table>

```

The output of the above code would be:

Personal Information	
Name	John Doe
Age	Skills
30	
HTML, CSS, JavaScript	

Exercise 23 Recipe Facts Table

Create a new section for recipe facts, inside the main element after the steps. It starts with an h2 heading with the text "Recipe Facts".

Working Space

```

1 <section>
2   <h2>Recipe Facts</h2>
3 </section>

```

Add a table that shows basic facts about the chocolate chip cookie recipe, such

as prep time, bake time, and servings. The table must have three rows and two columns. Add a border attribute to the `<table>` element with a value of "1" to make the table borders visible. (This attribute is old-fashioned; modern pages draw borders with CSS, but it still works.)

Prep Time	15 minutes
Bake Time	10 minutes
Servings	24 cookies

Answer on Page 123

After this exercise, the page should now look like the following:

Chocolate Chip Cookies

This recipe makes soft and sweet chocolate chip cookies.

- [Ingredients](#)
- [Steps](#)

Ingredients

- 2 1/4 cups flour
- 1 teaspoon baking soda
- 1 teaspoon salt
- 1 cup butter
- 3/4 cup sugar
- 3/4 cup brown sugar
- 2 eggs
- 2 cups chocolate chips

Steps

1. Preheat the oven to 375°F.
2. Mix the dry ingredients in one bowl.
3. Mix the butter, sugar, brown sugar, and eggs in another bowl.
4. Stir the dry ingredients into the wet ingredients.
5. Add the chocolate chips.
6. Scoop the dough onto a baking sheet.
7. Bake for 9 to 11 minutes.

Recipe Facts

Prep Time	15 minutes
Bake Time	10 minutes
Servings	24 cookies

Important Note: Make sure not to overbake the cookies, or they will become hard.

Figure 3.7: HTML Tables Exercise Output

3.14 Forms

Forms allow users to input and submit data. The `<form>` element contains form controls and is submitted to a server to process the provided data. The most common form controls include text fields, checkboxes, radio buttons, and submit buttons. These form controls are created using input elements.

Here is an example of a basic form structure with various input elements:

```
1 <form>
2   <label for="name">Name:</label>
3   <input type="text" id="name" name="name" required>
4
5   <label for="email">Email:</label>
6   <input type="email" id="email" name="email">
7
8   <input type="submit" value="Submit">
9 </form>
```

The `label` element is used to provide a label for form controls. The `for` attribute of the label should match the `id` of the corresponding input element to associate them together.

Input fields have various types, each serving a different purpose. The “`type`” attribute indicates the type of input field. Here is a list of the most common input types:

- `type="text"`: This is a single-line text input.
- `type="email"`: This is a single-line email input with validation that ensures the entered value is a valid email address.
- `type="password"`: This is a single-line password input (text is hidden).
- `type="number"`: This is a single-line numeric input.
- `type="checkbox"`: This is a checkbox for multiple selections.
- `type="radio"`: This is a radio button for single selection.
- `type="file"`: This is a file upload input.
- `type="date"`: This is a date picker input.
- `type="submit"`: This is a submit button.
- `type="reset"`: This is a reset button that will clear all form fields.

There are more form control elements besides the “`input`” element. Other form elements include:

- `<textarea>`: This is a multi-line text input.
- `<select>`: This is a dropdown menu. A dropdown is a list of options that can be expanded to show all available options.
- `<option>`: These are options within a dropdown.
- `<fieldset>`: This groups related form elements.

- `<legend>`: This is the title for a fieldset.

Here is an example that includes various form controls and uses fieldsets to group related information:

```
1 <form>
2   <fieldset>
3     <legend>Personal Information</legend>
4
5     <label for="username">Username:</label>
6     <input type="text" id="username" name="username">
7
8     <label for="password">Password:</label>
9     <input type="password" id="password" name="password">
10  </fieldset>
11
12  <fieldset>
13    <legend>Preferences</legend>
14
15    <label>
16      <input type="checkbox" name="newsletter">
17      Subscribe to newsletter
18    </label>
19
20    <label for="country">Country:</label>
21    <select id="country" name="country">
22      <option>Select a country</option>
23      <option value="usa">USA</option>
24      <option value="uk">UK</option>
25      <option value="ca">Canada</option>
26    </select>
27  </fieldset>
28
29  <button type="submit">Submit</button>
30 </form>
```

3.15 Images

Images are added using the `<img>` element, which is self-closing.

The basic syntax for the `<img>` element is:

```
1 
```

The most essential attributes for the `<img>` element are:

- **src:** The path to the image file. This is a required attribute that specifies the source of the image. The source path can route to an image file within the application or an external URL similar to a link. If the image cannot be found at the specified path, the browser will display a broken image icon. The **src** attribute is crucial for the image to load properly on the webpage. It is important to ensure that the path is correct and that the image file exists at that location.
- **alt:** This is alternative text displayed if the image cannot load or for screen readers. The **alt** attribute should provide a concise description of the image content.

Images come in various sizes. To control the display size of an image, you can use the width and height attributes.

- **width:** This specifies the width of the image in pixels or percentage.
- **height:** This specifies the height of the image in pixels or percentage.

Here is an example of using the width and height attributes to specify the dimensions of an image:

```
1 
```

For the next exercise, we will add an image to the cookie recipe page. Use the image below as a reference for the image you will add to the page.



Figure 3.8: Image of chocolate chip cookies

Exercise 24 The Recipe Image

Add a picture of the cookies to the bottom of the header element.

Working Space

```
1 <header>
2   <h1 id="recipe-title">Chocolate Chip
   ↳ Cookies</h1>
3   <p class="intro" title="A short recipe
   ↳ description">
4     This recipe makes soft and sweet
   ↳ chocolate chip cookies.
5   </p>
6
7   <!-- nav is here. Just showing the header
   ↳ for context -->
8
9   <!-- Add the image element here -->
10 </header>
```

Add an `<img>` element with the source set to "cookies.jpg". Make sure that the image file is in the same folder as your HTML file. The `alt` attribute should be set to "Fresh chocolate chip cookies on a plate". Set the width of the image to 500 pixels and the height to 300 pixels.

Answer on Page 124

After this exercise, the page should now look like the following:

Chocolate Chip Cookies

This recipe makes soft and sweet chocolate chip cookies.

- [Ingredients](#)
- [Steps](#)



Ingredients

- 2 1/4 cups flour
- 1 teaspoon baking soda
- 1 teaspoon salt
- 1 cup butter
- 3/4 cup sugar
- 3/4 cup brown sugar
- 2 eggs
- 2 cups chocolate chips

Steps

1. Preheat the oven to 375°F.
2. Mix the dry ingredients in one bowl.
3. Mix the butter, sugar, brown sugar, and eggs in another bowl.
4. Stir the dry ingredients into the wet ingredients.
5. Add the chocolate chips.
6. Scoop the dough onto a baking sheet.
7. Bake for 9 to 11 minutes.

Recipe Facts

Prep Time	15 minutes
Bake Time	10 minutes
Servings	24 cookies

Important Note: Make sure not to overbake the cookies, or they will become hard.

Figure 3.9: HTML Images Exercise Output

3.16 The Finished Page

Here is the whole recipe page, with every piece from this chapter's exercises in place. Save it as `index.html`, in the same folder as `cookies.jpg`, and open it in your browser.

1 `<!DOCTYPE html>`

```

2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Chocolate Chip Cookies</title>
7 </head>
8 <body>
9   <header>
10    <h1 id="recipe-title">Chocolate Chip Cookies</h1>
11    <p class="intro" title="A short recipe description">
12      This recipe makes soft and sweet chocolate chip cookies.
13    </p>
14
15    <nav>
16      <ul>
17        <li><a href="#ingredients">Ingredients</a></li>
18        <li><a href="#steps">Steps</a></li>
19      </ul>
20    </nav>
21
22    
24  </header>
25
26  <main>
27    <section class="ingredients-section">
28      <h2 id="ingredients">Ingredients</h2>
29      <ul>
30        <li>2 1/4 cups flour</li>
31        <li>1 teaspoon baking soda</li>
32        <li>1 teaspoon salt</li>
33        <li>1 cup butter</li>
34        <li>3/4 cup sugar</li>
35        <li>3/4 cup brown sugar</li>
36        <li>2 eggs</li>
37        <li>2 cups chocolate chips</li>
38      </ul>
39    </section>
40
41    <section class="steps-section">
42      <h2 id="steps">Steps</h2>
43      <ol>
44        <li>Preheat the oven to 375°F.</li>
45        <li>Mix the dry ingredients in one bowl.</li>
46        <li>Mix the butter, sugar, brown sugar, and eggs in another bowl.</li>
47        <li>Stir the dry ingredients into the wet ingredients.</li>
48        <li>Add the chocolate chips.</li>
49        <li>Scoop the dough onto a baking sheet.</li>
50        <li>Bake for 9 to 11 minutes.</li>
51      </ol>
52    </section>
53
54    <section>

```

```

54     <h2>Recipe Facts</h2>
55     <table border="1">
56         <caption>Chocolate Chip Cookie Recipe Facts</caption>
57         <tbody>
58             <tr>
59                 <td>Prep Time</td>
60                 <td>15 minutes</td>
61             </tr>
62             <tr>
63                 <td>Bake Time</td>
64                 <td>10 minutes</td>
65             </tr>
66             <tr>
67                 <td>Servings</td>
68                 <td>24 cookies</td>
69             </tr>
70         </tbody>
71     </table>
72 </section>
73 </main>
74
75 <footer>
76     <p><strong>Important Note:</strong> Make sure not to overbake the cookies, or
77     ↪ they will become hard.</p>
78 </footer>
79 </body>
</html>

```

The page has a clear structure, but it looks plain: default fonts, blue underlined links, and a table with clunky borders. It also cannot do anything except scroll. In the CSS and JavaScript chapter, you will give it colors and a layout that works on a phone, add a dark mode button and a box that scales the recipe, and serve it from an Express server with a like button. Keep this file; it is where that chapter begins.

Web APIs

4.1 Making Web Requests with HTTP

When you open a weather app, it does not know the forecast. It sends a short message across the internet to a server, asking “What is the weather in Denver?”, and the server sends back an answer. Your phone does this hundreds of times a day. The How the Internet Works chapter showed how those messages travel as packets; in this chapter, you will learn the language the messages are written in, and then build a server of your own.

The Hypertext Transfer Protocol (HTTP) is the protocol used for transmitting hypertext over the World Wide Web. It is the foundation of any data exchange on the web, and it is a protocol used for transmitting hypertext requests from clients (like a user’s browser) to servers, which respond with the requested resources.

4.1.1 HTTP Requests

An HTTP request is made up of several components:

- **Method:** The HTTP method, like GET (retrieve data), POST (send data), PUT (update data), DELETE (remove data), and so on.
- **URL:** The URL of the resource to retrieve, send data to, update or delete.
- **Headers:** Additional information about the request or response, like the content type of the body.
- **Body:** The body of the request, used when sending data in POST or PUT requests.

One way to think about an HTTP request is to imagine ordering food at a restaurant. The method is the type of action you want to take, such as asking to see the menu, placing an order, changing an order, or canceling it. The URL is like telling the server exactly which restaurant counter to go to. The headers are like the extra details you give, such as saying you want vegetarian food or that you would like water with your meal. The body is the actual order information, such as the list of food you want to buy.

The method tells the server what kind of action the client wants to perform. For example, a GET request is used when the client wants to receive information from the server, such as loading a web page or asking for a list of items. A POST request is used when the client wants to send new information to the server, such as creating an account or submitting

a form. A PUT request is used to update something that already exists, and a DELETE request is used to remove something.

The URL works like an address. It tells the request where to go. If the method explains what the client wants to do, the URL explains where that action should happen. For example, if a web app wants a list of users, it might send a request to a URL that ends in `/users`. If it wants information for one specific user, it might send a request to a URL like `/users/5`.

Headers help the client and server understand each other better. They carry extra information along with the request or response, such as what type of data is being sent. This is similar to writing “Fragile” on a package or adding a note that says “Please handle with care.” The message is still the same package, but the note helps explain how it should be handled.

The body is the part of the request that contains the actual data being sent. For example, when a user fills out a sign-up form, the name, email address, and password they enter may be placed in the body of the request. Not every request needs a body. GET requests usually do not send one, because they are mainly asking to receive information rather than send it.

4.1.2 Using HTTP with Web-Based APIs

Software developers often use HTTP to interact with web-based APIs. An Application Programming Interface (API) is a set of rules that allows programs to talk to each other. The developer creates the API on the server and allows the client to talk to it.

When a developer makes a request to an API endpoint, they are asking the server to either send them some data or receive some data from them. The response from the server will often be in a format like JSON or XML, which the developer can then use in their own application.

For example, a developer might make a GET request to `https://api.example.com/users` to retrieve a list of all users. The server would respond with a list of users in a format like JSON.

An API endpoint is a specific URL that a client can use to communicate with a server. Each endpoint usually has a clear purpose. One endpoint might return a list of books, while another might return information about a single book. In the same way that a restaurant has different counters for ordering food, paying, or picking up an order, an API can have different endpoints for different tasks.

When a client sends a request to an API, the server reads the request and decides how to respond. If the request is valid, the server sends back a response with the requested data

or a message saying the action worked. If something is wrong, the server sends back a response explaining that there was a problem. This back-and-forth communication is one of the main ways web applications work behind the scenes.

For example, imagine a school website that stores a list of students. A client could send a GET request to an endpoint like `/students` to ask for the full list. It could send a POST request to the same endpoint to add a new student. It could send a PUT request to `/students/3` to update the information for one student, or a DELETE request to `/students/3` to remove that student from the list. These requests follow the same HTTP ideas, even if the website or app itself looks very different on the screen.

Many modern apps work this way. A weather app may send a request to get the latest forecast. A music app may send a request to get a list of songs. A shopping app may send a request to place an order. Even though users may only see buttons, menus, and pictures, HTTP requests are often moving data in the background.

4.1.3 HTTP Responses

After a server receives an HTTP request, it sends back an HTTP response. The response is the server's answer to the client. Just as a request has several parts, a response also has its own important parts.

An HTTP response usually includes a status code, headers, and sometimes a body. The status code tells the client what happened. The headers give extra information about the response. The body contains the returned data, if there is any.

A simple way to think about a response is to imagine asking a librarian for a book. The librarian might say, "Here is the book you asked for," "That book is checked out," or "We do not have that book." These are different kinds of responses to the same kind of request.

Some common HTTP status codes are:

- 200: The request worked, and the server is sending back the requested data.
- 201: The request worked, and the server created something new.
- 204: The request worked, and there is nothing to send back (common after a delete).
- 401: The client has not proven who it is (for example, a missing API key).
- 400: The request had a problem, such as missing or incorrect information.
- 404: The server could not find what was requested.
- 500: Something went wrong on the server.

These codes help the client understand the result quickly. Instead of reading a long message first, the client can look at the status code and get a fast summary of what happened.

4.1.4 Data Formats in Responses

When a server sends data back to a client, that data needs to be organized in a way both sides can understand. Two common formats are JSON and XML. These formats are simply ways of arranging information so that programs can read it clearly and consistently.

JSON is one of the most common formats used with web APIs. It is popular because it is lightweight and easy for both humans and programs to read. A JSON response might include data written as key–value pairs. For example, a user might be described with data such as a name, age, and email address.

Here is a small example of JSON:

```
{
  "name": "Ava",
  "grade": 10,
  "club": "Robotics"
}
```

In this example, "name", "grade", and "club" are the labels, and "Ava", 10, and "Robotics" are the values. This is similar to filling out an information card. Each label tells what kind of information is being stored, and each value is the actual information.

XML is another format that may be used. It organizes information with opening and closing tags. While it is less common in many modern beginner examples, it is still useful to know that different APIs may return data in different formats.

4.2 Calling a Real API from Python

Enough theory: let us ask a real server for real data. Open-Meteo is a free weather service with a public API that needs no key. You will send it a GET request from Python and get today's weather back as JSON.

Python's requests library makes HTTP requests easy. Install it the same way you installed matplotlib:

```
1 pip install requests
```

Then create a file called `weather.py`:

```
1 import requests
2
3 url = "https://api.open-meteo.com/v1/forecast"
4 params = {
5     "latitude": 39.74,          # Denver
6     "longitude": -104.99,
7     "current": "temperature_2m,wind_speed_10m",
8     "temperature_unit": "fahrenheit",
9     "wind_speed_unit": "mph",
10 }
11
12 response = requests.get(url, params=params)
13 print("Status code:", response.status_code)
14 print("Full URL:", response.url)
15
16 data = response.json()          # turn the JSON text into a Python dictionary
17 now = data["current"]
18 print("Time (UTC):", now["time"])
19 print("Temperature:", now["temperature_2m"], "°F")
20 print("Wind speed:", now["wind_speed_10m"], "mph")
```

Here is what is happening:

- `requests.get` sends a GET request. The `params` dictionary becomes the *query string*, the part of the URL after the `?`, with `&` between each `name=value` pair. The program prints the full URL so you can see it. (The `%2C` is how a comma is written in a URL.)
- `response.status_code` is the HTTP status code. 200 means it worked.
- `response.json()` turns the JSON in the response body into a Python dictionary. JSON objects become dictionaries, and JSON lists become Python lists, so you reach into the data with square brackets, just as you would with any dictionary.

When you run it, you should see something like this (your numbers will be different, because the weather changes):

```
Status code: 200
Full URL: https://api.open-meteo.com/v1/forecast?latitude=39.74&longitude=-104.99&current=temperature_2m,wind_speed_10m
Time (UTC): 2026-10-22T15:00
Temperature: 59.4 °F
Wind speed: 3.2 mph
```

You can paste the full URL into a web browser, too. The browser sends the same GET request and shows you the raw JSON that Python received. Every weather app on every phone is doing something very much like this, many times a day.

Exercise 25 Which Method?

Which HTTP method would an app most likely use for each action?

1. Showing your profile page
2. Creating a new account
3. Changing only your profile picture
4. Deleting a post

Working Space

Answer on Page 124

Exercise 26 Reading the JSON

Open-Meteo sends back JSON that includes this:

```
"current": {"time": "2026-10-22T15:00", "temperature_2m": 59.4}
```

After `data = response.json()`, what Python expression gives the temperature? What type of Python value is it?

Working Space

Answer on Page 125

Exercise 27 Your Town

Look up the latitude and longitude of your town (a map app will show them), and change `weather.py` to print your

Working Space

own weather.

Answer on Page 125

Exercise 28 A Three-Day Forecast

Open-Meteo can also send a daily forecast. Change the parameters to

Working Space

```
1 params = {  
2     "latitude": 39.74,  
3     "longitude": -104.99,  
4     "daily":  
5         ↪ "temperature_2m_max,temperature_2m_min",  
6     "temperature_unit": "fahrenheit",  
7     "timezone": "America/Denver",  
8     "forecast_days": 3,  
9 }
```

and print the date, high, and low for each of the three days. (Print `data["daily"]` first to see how the data is organized.)

Answer on Page 125

Exercise 29 A Bad Request

Change the latitude to 139.74 and run `weather.py` again. What status code do you get? Print `response.json()` to see why. Why is this a 400 and not a 404?

Working Space

Answer on Page 125

In the next chapter, Building a Web API, you will switch sides and write a server of your own.

Building a Web API

In the Web APIs chapter, you were the client, asking a weather server for data. Now you will switch sides and build an API of your own. It will manage a list of students, and by the end of the chapter it will support every CRUD operation, require an API key, and be testable from both the command line (curl) and a graphical tool (Postman).

5.1 Setting up an API

Create a file called `server.js`. This will be a basic webserver that we will use to test our API. This is called a *RESTful API*. Write the following code to create a basic server for our API.

```
1  const express = require("express");
2
3  const app = express();
4  const PORT = 3000;
5
6  const student = {
7    student_id: 42,
8    name: "John Smith",
9    major: "Computer Science",
10   credits_completed: 88
11 };
12
13 app.get("/students/42", (req, res) => {
14   res.json(student);
15 });
16
17 app.listen(PORT, () => {
18   console.log(`API running at http://localhost:${PORT}`);
19 });
```

Then, run the following

```
1  npm init -y
2  npm install express
3  node server.js
```

This will create a new server, along with a ton of dependency files that javascript and

HTTP Method	Purpose	CRUD Op	Example
GET	Fetch Data	READ	Get student information
PUT	Update Data	UPDATE	Fully update student
PATCH	Update section of Data	UPDATE	Update a student's major
POST	Create Data	CREATE	Create a new student
DELETE	Remove Data	DELETE	Remove a student after graduation

node automatically install as part of your server.

5.1.1 What is an endpoint?

An *endpoint* is a URL exposed by the API. It usually is able to display a page or data based on the request of the endpoint.

We can navigate to the endpoint `GET /students/42`, where `students` is the resource that usually maps to a database, and `42` is the id of a student in the database. To retrieve this student from the database, we can navigate to

```
1 GET http://localhost:3000/students/42
```

which is an endpoint from our API. This will output

```
1 {  
2   "student_id": 42,  
3   "name": "John Smith",  
4   "major": "Computer Science",  
5   "credits_completed": 88  
6 }
```

Note that this is done in json format, common for backend data parsing.

Each endpoint combines a resource path with an HTTP method to complete an action.

This leads us to the idea of *CRUD operations*. *CRUD*, which stands for *CREATE*, *READ*, *UPDATE*, and *DELETE*, are a form of outlining an APIs routes. Calling a route that aims to alter should follow one of the *CRUD* operations. *GET* is the *READ* operation, since it only reads data.

For a database representing students (assume more than just one), we would want to have routes that can complete each *CRUD* operation.

Before an API can receive data from a client, Express must parse JSON request bodies.

Add the following line after creating the application:

```
1 app.use(express.json());
```

This middleware automatically converts incoming JSON into a JavaScript object that can be accessed through `req.body`.

5.1.2 Reading all resources

Most APIs provide a route for retrieving an entire collection. Change the `students` array to look like the following.

```
1 const students = [  
2   {  
3     student_id: 42,  
4     name: "John Smith",  
5     major: "Computer Science",  
6     credits_completed: 88  
7   },  
8   {  
9     student_id: 43,  
10    name: "Jane Doe",  
11    major: "Mathematics",  
12    credits_completed: 61  
13  }  
14 ];  
15  
16 app.get("/students", (req, res) => {  
17   res.json(students);  
18 });
```

The endpoint

```
1 GET /students
```

returns every student currently stored in the collection. Notice that we did not include an `id`. Now, there are more than one possible `ids`, so we need to make a general route that takes in an `id` of a student to fetch

```
1 app.get("/students/:id", (req, res) => {  
2   // use the id that is passed in the route  
3   const id = Number(req.params.id);  
4   // find a student from our students array
```

```

5  const student = students.find((s) => s.student_id === id);
6
7  // if the id is not found, we assume the student is not in the array
8  if (!student) {
9    return res.status(404).json({ error: "Student not found" });
10 }
11
12 res.json(student);
13 });

```

Start the server. Now, `http://localhost:3000/students` returns the entire students array, while `http://localhost:3000/students/43` only returns information about Jane Doe.

Routing to `http://localhost:3000/students/19` is invalid, so it returns

```

1  {"error": "Student not found"}

```

5.2 Requests with cURL

Curl, sometimes stylized as cURL or curl, is a command-line based resource used for sending API and HTTP requests for web server data.

The simplest curl command

```

1  curl http://example.com

```

which will return the html source code of the website <http://example.com>.

```

1  <!doctype html>
2  <html lang="en">
3
4  <head>
5    <title>Example Domain</title>
6    <link rel="icon" href="data:,">
7    <meta name="viewport" content="width=device-width, initial-scale=1">
8    <style>
9      body {
10        background: #eee;
11        width: 60vw;
12        margin: 15vh auto;
13        font-family: system-ui, sans-serif
14      }
15
16      h1 {
17        font-size: 1.5em

```

```
18     }
19
20     div {
21       opacity: 0.8
22     }
23
24     a:link,
25     a:visited {
26       color: #348
27     }
28   </style>
29 </head>
30
31 <body>
32   <div>
33     <h1>Example Domain</h1>
34     <p>This domain is for use in documentation examples without needing
35     ↪ permission. Avoid use in operations.</p>
36     <p><a href="https://iana.org/domains/example">Learn more</a></p>
37   </div>
38 </body>
39 </html>
```

5.3 Creating resources with POST

So far, curl has only read data. It can also send data. Going back to our student API, let us add a new student.

A POST request creates a new resource. Unlike GET requests, POST requests send data inside the request body.

```
1 app.post("/students", (req, res) => {
2   const student = req.body;
3
4   // basic validation: ensure the body has at least one field
5   if (!student || Object.keys(student).length === 0) {
6     return res.status(400).json({ error: "Invalid student data" });
7   }
8
9   students.push(student);
10
11   // status 201 means that the student was pushed successfully
12   res.status(201).json(student);
13 });
```

Notice the use of

```
1 req.body
```

which contains the JSON sent by the client.

HTTP requests are composed of headers and a body. Headers are metadata such as Content-Type (most commonly application/json) or Authorization that describe how the request should be handled. The body is the payload that carries actual data, like the student record being created by a POST request.

In Express, request headers are available on `req.headers`. For JSON payloads, middleware like `express.json()` parses the body so that the data is available on `req.body`.

A template for sending json in the body looks something like the following:

```
1 curl -X METHOD URL \  
2     -H "Header: Value" \  
3     -d 'JSON DATA'
```

To send a request body, include the body data in the HTTP request and set the appropriate Content-Type header. For example, with curl:

```
1 curl -X POST http://localhost:3000/students \  
2     -H "Content-Type: application/json" \  
3     -d '{  
4         "student_id": 44,  
5         "name": "Jerry Doe",  
6         "major": "Computer Science",  
7         "credits_completed": 23  
8     }'
```

This can be broken down as:

- `curl`: The command-line tool for making HTTP requests.
- `-X POST`: Use the POST HTTP method.
- `http://localhost:3000/students`: The endpoint being contacted. Must exist in our server routes.
- `-H "Content-Type: application/json"`: Tell the server the request body contains JSON.
- `-d '{ ... }'`: Send this JSON as the request body.

The request body in this example is the JSON string passed with `-d`. Express will parse

that JSON into `req.body` when `express.json()` middleware is enabled.

The POST handler adds Jerry Doe to the `students` array in memory, so a GET request on `/students` now returns three students. This data lives only while the server runs. It is not saved anywhere, so restarting the server resets the list.

```
1  [
2    {
3      "student_id": 42,
4      "name": "John Smith",
5      "major": "Computer Science",
6      "credits_completed": 88
7    },
8    {
9      "student_id": 43,
10     "name": "Jane Doe",
11     "major": "Mathematics",
12     "credits_completed": 61
13   },
14   {
15     "student_id": 44,
16     "name": "Jerry Doe",
17     "major": "Computer Science",
18     "credits_completed": 23
19   }
20 ]
```

5.4 Updating and Deleting Resources with cURL

Our API can now read students and create new ones. To finish out the CRUD table from earlier, we need routes for PUT, PATCH, and DELETE, and we need to know how to trigger each of them with curl.

5.4.1 Fully replacing a resource with PUT

A PUT request means *replace this entire resource with what I am sending you*. If you leave out a field, that field is expected to disappear or reset, because the whole object is being overwritten.

```
1  app.put("/students/:id", (req, res) => {
2    const id = Number(req.params.id);
3    const index = students.findIndex((s) => s.student_id === id);
4
5    if (index === -1) {
6      return res.status(404).json({ error: "Student not found" });
7    }
8  })
```

```

7   }
8
9   // PUT replaces the whole object, so we overwrite it completely
10  students[index] = {
11    student_id: id, ...req.body // this format keeps the same student_id and
    ↳ updates the rest of the fields with the new data
12  };
13  res.json(students[index]);
14  });

```

```

1  curl -X PUT http://localhost:3000/students/43 \
2    -H "Content-Type: application/json" \
3    -d '{
4      "name": "Jane Doe",
5      "major": "Physics",
6      "credits_completed": 64
7    }'

```

Notice we did not send `student_id` in the body – the route pulls it from the URL instead. Try leaving out `credits_completed` entirely and re-running the request. Because PUT replaces the whole object, that field will not be included in the response. This is the behavior that distinguishes PUT from PATCH.

5.4.2 Partially updating a resource with PATCH

A PATCH request means *only change the fields I am sending you, and keep everything else the same*. This is almost always what you want when, say, updating a single field like a student’s major.

```

1  app.patch("/students/:id", (req, res) => {
2    const id = Number(req.params.id);
3    const student = students.find((s) => s.student_id === id);
4
5    if (!student) {
6      return res.status(404).json({ error: "Student not found" });
7    }
8
9    // PATCH merges the incoming fields onto the existing object
10   Object.assign(student, req.body);
11   res.json(student);
12 });

```

```

1  curl -X PATCH http://localhost:3000/students/43 \
2    -H "Content-Type: application/json" \

```

```
3 -d '{ "major": "Physics" }'
```

Run a GET on `/students/43` afterward and confirm that `name` and `credits_completed` are untouched; only `major` changed. That single-field behavior is the whole point of PATCH.

5.4.3 Removing a resource with DELETE

A DELETE request removes a resource. Unlike POST, PUT, and PATCH, it typically does not need a request body; the URL alone tells the server what to remove.

```
1 app.delete("/students/:id", (req, res) => {  
2   const id = Number(req.params.id);  
3   const index = students.findIndex((s) => s.student_id === id);  
4  
5   if (index === -1) {  
6     return res.status(404).json({ error: "Student not found" });  
7   }  
8  
9   students.splice(index, 1);  
10  // status 204 means "success, but there is nothing to send back"  
11  res.status(204).send();  
12 });
```

```
1 curl -X DELETE http://localhost:3000/students/44
```

This removes Jerry Doe (student 44), whom you added with POST. Then run GET `/students` again and Jerry Doe should be gone. If you run the same DELETE command a second time, you should get a 404, since the student no longer exists to delete.

5.4.4 Headers with `-i` and `-v`

Every curl command so far has only shown you the body of the response. But remember: every HTTP exchange also includes headers and a status code, even when curl hides them by default. Two flags make that conversation visible.

`-i` includes the response headers above the body. The output below leaves out a few headers that Express adds, such as `Date` and `ETag`:

```
1 curl -i http://localhost:3000/students/43
```

```
1 HTTP/1.1 200 OK
2 Content-Type: application/json; charset=utf-8
3 Content-Length: 76
4
5 {"student_id":43,"name":"Jane Doe","major":"Physics","credits_completed":64}
```

-v (verbose) goes further and shows the *request* curl sent, not just the response it received
– useful when a request is failing and you are not sure what curl is actually sending:

```
1 curl -v -X PATCH http://localhost:3000/students/43 \
2   -H "Content-Type: application/json" \
3   -d '{ "major": "Physics" }'
```

Reading a -v output line by line is one of the best habits a beginner can build: lines starting with > are what curl sent, lines starting with < are what the server sent back.

5.5 Authentication and API Keys

Every route we have built so far is completely open: anyone who knows the URL can read, create, update, or delete students. Real APIs usually cannot work that way, since that data might be private, or changing it might cost money (imagine an API that sends text messages or charges a credit card). To restrict access, APIs commonly require an *API key*: a long, unique string that identifies who is making the request.

An API key is typically sent as a request header, most often Authorization or a custom header like X-API-Key:

```
1 curl http://localhost:3000/students \
2   -H "Authorization: Bearer abc123yourkeyhere"
```

On the server, you check for that header before doing anything else:

```
1 const API_KEY = "abc123yourkeyhere";
2
3 function requireApiKey(req, res, next) {
4   const authHeader = req.headers.authorization;
5
6   if (authHeader !== `Bearer ${API_KEY}`) {
7     return res.status(401).json({ error: "Unauthorized" });
8   }
9 }
```

```
10   next();
11 }
12
13 app.get("/students", requireApiKey, (req, res) => {
14   res.json(students);
15 });
```

If the header is missing or wrong, the client gets back a 401 Unauthorized instead of the data.

A few rules are worth internalizing early, because they matter far beyond this textbook:

- **Never commit an API key to a public repository.** Once a key is pushed to a code repository such as GitHub, assume it is compromised, even if you delete it in a later commit. If you do accidentally commit a key, revoke it immediately and generate a new one.
- **Never hardcode a real key directly in your source code.** The example does for simplicity. Real projects store keys in environment variables (often loaded from a `.env` file that is excluded from version control or public repositories) and read them with something like `process.env.API_KEY`.
- **Treat an API key like a password.** Anyone who has your key can act as you. They could be reading your data, spending your quota, or in some APIs, spending your money.
- **You can only access APIs with their specific key.** If you have multiple APIs, each one will have its own key. You cannot use the same key for different APIs. For example, Spotify's public API has its own key, and a method for you to generate your own key. You cannot use a key from another API (such as OpenAI) to access Spotify's API. The strength here is that if one key is compromised, it does not compromise all of your APIs. Secondly, this way, if a breach of data is attempted, Spotify knows who owns the key and can revoke it. Feel free to learn more about Spotify's API and how to generate your own key at <https://developer.spotify.com/documentation/web-api/>. Becoming increasingly more common is OpenAI's API, which has its own key and method for generating your own key. You can learn more about OpenAI's API at <https://platform.openai.com/docs/guides/authentication>.

This is the difference between *authentication* (proving *who* you are, usually with a key or token) and *authorization* (determining *what* you are allowed to do once you have proven who you are). A key can identify you correctly and still be denied access to a specific route if you do not have permission for it.

cURL	Postman
The URL itself	The URL bar
-X METHOD	The method dropdown
-H "Header: Value"	The Headers tab
-d '{...}'	The Body tab (raw / JSON)
-i / -v output	The Response panel and Console

Exercise 30 Changing One Field with PATCH

Use curl and a PATCH request to change Jane Doe's `credits_completed` to 70. Leave her other fields alone. Then send a GET request to `/students/43` and check the result.

Working Space

Answer on Page 125

5.6 Postman

Typing curl commands by hand is a great way to *learn* how HTTP requests actually work, because every piece of the request (the method, the headers, the body) has to be spelled out yourself. But once you already understand what those pieces are, typing them out every single time gets tedious and error-prone, especially for requests with long JSON bodies or several headers.

Postman is a GUI application for building, sending, and saving HTTP requests. Everything you have been typing as curl flags has a matching box in Postman:

5.6.1 Building your first request in Postman

1. Open Postman (download available [here](#)) and click **New** → **HTTP Request**.
2. Set the method dropdown (left of the URL bar) to GET, and enter `http://localhost:3000/students` in the URL bar.
3. Click **Send**. The response body appears in the panel below, along with the status code and response time: the same information `-i` shows you in curl, just laid out visually instead of printed as text.

Now recreate the POST request from earlier:

1. Change the method to POST and set the URL to `http://localhost:3000/students`.
2. Open the **Body** tab, select **raw**, and choose **JSON** from the format dropdown on the right. Postman will automatically set the `Content-Type: application/json` header for you. This is one thing it does that curl makes you do by hand.
3. Paste in a student object:

```
1 {  
2   "student_id": 45,  
3   "name": "Alex Rivera",  
4   "major": "Biology",  
5   "credits_completed": 12  
6 }
```

4. Click **Send** and confirm you get a 201 Created status back with the same object.

5.6.2 Why this matters once requests get complicated

A single GET request is barely worth using Postman for; a browser address bar does the job just as well. Postman starts to pay off once your API involves things that are painful to keep retyping in a terminal:

- **Collections.** Every request you build can be saved into a named *collection*, so your entire API (every route in this chapter, for example) can be saved, organized, and rerun with a click instead of retyped.
- **Environments.** A variable like `{{base_url}}` can be swapped between `http://localhost:3000` and a real deployed server without editing every saved request individually.
- **Authorization tab.** Instead of manually adding an Authorization header on every request, you can set your API key once per collection and Postman attaches it automatically.
- **Automated testing:** Postman can run a series of requests and check that the responses match expected values, so you can verify your API is working as intended after every change. This is great for testing before you have a front-end built, or for running regression tests after a code change.

Neither tool replaces the other. cURL is scriptable and works anywhere you have a terminal, including inside automated tests and shell scripts, which is why you will keep seeing it later in this book. Postman is better suited to exploring an API interactively, organizing a large set of requests, and sharing that organized set with teammates. Most professional API developers end up using both, for different parts of their workflow.

5.7 More Practice

Exercise 31 Predict the Status Code

For the student server in this chapter, predict the status code of each request.

1. GET /students/42
2. GET /students/19
3. POST /students with a complete new student in the body
4. DELETE /students/44, run twice in a row (after student 44 was added)
5. GET /students without the right API key, once requireApiKey is in place

Working Space

Answer on Page 126

Exercise 32 Filtering by Major

A client wants only the math majors. Change the GET /students route so that GET /students?major=Mathematics returns only the students whose major matches, while plain GET /students still returns everyone. (Hint: Express puts the query string in req.query, so req.query.major is "Mathematics".)

Working Space

Answer on Page 126

Exercise 33 Better Validation

The POST route accepts any body with

Working Space

at least one field, so a client could add a student with no name or with -5 credits. Change it to reply with `400` and a helpful error message unless `name` is a non-empty string and `credits_completed` is a number that is 0 or more.

Answer on Page 126

5.8 Summary

This chapter touched on a lot of software and development concepts. Here are some of the key takeaways:

- APIs are ways to connect and fetch data from a server. They are a way to connect two different applications together.
- An HTTP exchange is a request (method, URL, headers, body) and a response (status code, headers, body).
- APIs are usually built with a RESTful architecture, which is a set of principles that follow HTTP protocol to perform CRUD operations on resources.
- **CURL** is a command line tool that allows you to send HTTP requests to an API. Another commonly used tool is **Postman**, which is a GUI application that allows you to send HTTP requests to an API.
- We created a basic API that allows us to perform CRUD operations on a student resource. We can create, read, update, and delete students from our API.

CHAPTER 6

CSS and JavaScript

At the end of the HTML chapter, you had a complete recipe page for chocolate chip cookies. It works, but it looks like a web page from 1995: black Times New Roman on white, blue underlined links, and a table with clunky double borders. Nobody would send that to a friend.

In this chapter, you will turn that page into something you would be proud to show off. You will use *CSS* to give it colors, fonts, rounded buttons, and a layout that works on both a laptop and a phone. Then you will use *JavaScript* to make it respond to clicks: a dark mode button, steps you can cross off as you bake, and a box that doubles or triples the recipe. Finally, you will serve the page from your own Express server, like the one you built in the Building a Web API chapter, and add a like button that counts likes from everyone who visits.

Chocolate Chip Cookies

This recipe makes soft and sweet chocolate chip cookies.

- [Ingredients](#)
- [Steps](#)



Ingredients

- 2 1/4 cups flour
- 1 teaspoon baking soda
- 1 teaspoon salt
- 1 cup butter
- 3/4 cup sugar
- 3/4 cup brown sugar
- 2 eggs
- 2 cups chocolate chips

Steps

Chocolate Chip Cookies

This recipe makes soft and sweet chocolate chip cookies.

Ingredients Steps



Batches: Dark mode  12

Ingredients

- 4 1/2 cups flour
- 2 teaspoon baking soda
- 2 teaspoon salt
- 2 cup butter

Steps

1. Preheat the oven to 375°F.
2. Mix the dry ingredients in one bowl:
3. Mix the butter, sugar, brown sugar, and eggs in another bowl.

Figure 6.1: The recipe page at the end of the HTML chapter (left) and at the end of this chapter (right), with the recipe doubled and the first two steps crossed off.

6.1 Setting Up

Make a folder called `recipe`, and inside it, a folder called `public`. Into `public`, put the finished page from the end of the HTML chapter, saved as `index.html`, along with `cookies.jpg`. By the end of the chapter, the project will look like this:

`recipe/`

server.js	the Express server
public/	
index.html	the structure (HTML)
style.css	the looks (CSS)
script.js	the behavior (JavaScript)
cookies.jpg	

Keeping the three languages in three files is a good habit. When something looks wrong, you know to look in `style.css`; when a button does nothing, you know to look in `script.js`.

Until the last section, you do not need a server. Just double-click `index.html` to open it in your browser, and press reload each time you save a change.

6.2 Your First Stylesheet

A CSS file is a list of *rules*. Each rule picks some elements with a *selector*, then sets *properties* on them:

```
1 selector {
2   property: value;
3   another-property: value;
4 }
```

Create `style.css` in the public folder, and start with these rules:

```
1 body {
2   font-family: system-ui, sans-serif;
3   line-height: 1.6;
4   color: #333333;
5   background-color: #fffaf3;
6   max-width: 800px;
7   margin: 0 auto;
8   padding: 1rem;
9 }
10
11 h1, h2 {
12   color: #7b3f00;
13 }
14
15 .intro {
16   font-size: 1.2rem;
17   font-style: italic;
18 }
```

To connect the stylesheet to the page, add a `<link>` element to the `<head>` of `index.html`:

```
1 <link rel="stylesheet" href="style.css">
```

Reload the page. The text switches to your computer's own interface font (`system-ui`), the background turns a warm cream, the headings turn chocolate brown, and the whole page sits in a centered column instead of stretching across a wide screen.

A few things to notice:

- **Colors** are hex codes, just like the ones in the Data Representation chapter: `#7b3f00` means red 7b, green 3f, blue 00. CSS also understands about 140 color names, such as `white` and `tomato`.
- **Sizes** come in units. `px` is a pixel. `rem` is the height of the page's normal text, so `1.2rem` is 20% bigger than normal. `line-height: 1.6` (no unit) means each line takes 1.6 times the height of the text, which makes paragraphs easier to read.
- `margin: 0 auto` sets the top and bottom margins to 0 and lets the browser split the leftover space equally on the left and right. That is the classic trick for centering a column.

You may remember the `style` attribute from the HTML chapter, which puts CSS directly on one element. It works, but a separate stylesheet lets one rule restyle every page on your site at once.

6.3 Selectors and the Cascade

Selectors are how CSS finds elements. These are the ones you will use most:

Selector	Picks
<code>h2</code>	every <code><h2></code> element
<code>.intro</code>	every element with <code>class="intro"</code>
<code>#batches</code>	the one element with <code>id="batches"</code>
<code>h1, h2</code>	every <code><h1></code> and every <code><h2></code>
<code>nav a</code>	every <code><a></code> inside a <code><nav></code>
<code>a:hover</code>	a link while the mouse is over it

Now you can see why the HTML chapter had you give elements `class` and `id` attributes: they are handles for CSS and JavaScript to grab.

What if two rules disagree? The name “Cascading Style Sheets” refers to the answer. The

more *specific* selector wins: an id beats a class, and a class beats a plain element name. If two selectors are equally specific, the one that comes later in the file wins.

6.4 The Box Model

To CSS, every element is a rectangle. Around its content are three layers: *padding* (space inside the border), the *border* itself, and *margin* (space outside the border, pushing other elements away).

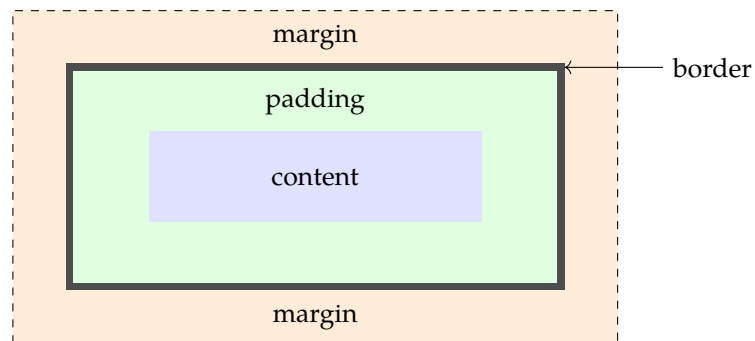


Figure 6.2: The box model. A background color fills the content and padding; the margin is always transparent.

Two rules from our stylesheet use the box model to fix the clunky table and the photo:

```

1  img {
2    max-width: 100%;
3    height: auto;
4    border-radius: 12px;
5  }
6
7  table {
8    border-collapse: collapse;
9  }
10
11 td {
12   border: 1px solid #d2b48c;
13   padding: 0.4rem 0.8rem;
14 }

```

`border: 1px solid #d2b48c` gives each cell a thin tan line, and `border-collapse` merges neighboring cells' borders into one. Now you can delete the old-fashioned `border="1"` attribute from the `<table>` tag. The padding gives the words in each cell room to breathe.

The image rule matters on phones. The `<img>` tag asks for a width of 500 pixels, which is wider than many phone screens. `max-width: 100%` says “never wider than the space

you are in,” and `height: auto` keeps the photo’s proportions. `border-radius` rounds the corners.

6.5 Layout with Flexbox

By default, elements stack from top to bottom. To line things up side by side, use *flexbox*. Setting `display: flex` on a container lays its children out in a row.

Start with the navigation links, which are currently a bulleted list. These rules turn them into a row of rounded buttons:

```

1  nav ul {
2      display: flex;
3      gap: 1rem;
4      list-style: none;
5      padding: 0;
6  }
7
8  nav a {
9      background-color: #7b3f00;
10     color: white;
11     text-decoration: none;
12     padding: 0.4rem 1rem;
13     border-radius: 999px;
14 }
15
16 nav a:hover {
17     background-color: #a0522d;
18 }

```

`list-style: none` removes the bullets, `text-decoration: none` removes the underlines, and a huge `border-radius` turns each rectangle into a pill. The `:hover` rule makes a button lighten when the mouse is over it, which tells the user it is clickable.

Next, put the ingredients and steps side by side, but only when there is room:

```

1  main {
2      display: flex;
3      flex-wrap: wrap;
4      gap: 2rem;
5  }
6
7  main section {
8      flex: 1 1 250px;
9  }

```

`flex: 1 1 250px` says each section would like to be 250 pixels wide, may grow to share extra space, and may shrink. `flex-wrap: wrap` lets sections that do not fit move to the next row. On a laptop, ingredients and steps sit next to each other; on a phone, they stack. A page that rearranges itself for the screen it is on is called *responsive*. (It also needs the viewport meta tag from the HTML chapter, which tells phones not to pretend to be a wide screen.)

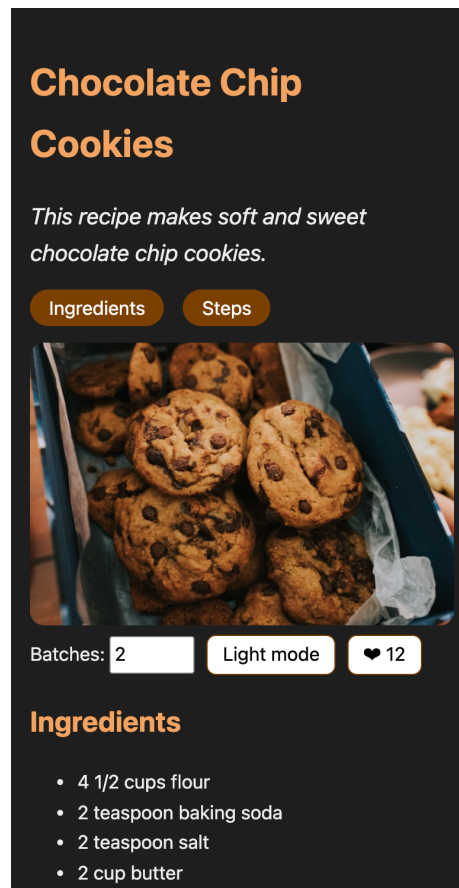


Figure 6.3: The same page on a phone, in dark mode. The ingredients and steps now stack.

6.6 JavaScript for Python Programmers

CSS controls how the page looks. To make it *do* things, you need a programming language that runs inside the browser, and there is only one: JavaScript. You have already seen some in the Building a Web API chapter, where your Express server was written in it. Since you know Python (and some C), most of JavaScript will look familiar:

Python	JavaScript
<code>x = 5</code>	<code>let x = 5;</code> <code>const y = 10; (cannot be reassigned)</code>
<code>print("hi", x)</code>	<code>console.log("hi", x);</code>
<code>if x > 3:</code>	<code>if (x > 3) { ... }</code>
<code>elif, else</code>	<code>else if, else</code>
<code>for item in items:</code>	<code>for (const item of items) { ... }</code>
<code>def area(r):</code>	<code>function area(r) { ... }</code>
<code>[1, 2, 3] (list)</code>	<code>[1, 2, 3] (array)</code>
<code>{"likes": 3} (dict)</code>	<code>{ likes: 3 } (object)</code>
<code>and, or, not</code>	<code>&&, , !</code>
<code>==</code>	<code>===</code>
<code>True, False, None</code>	<code>true, false, null</code>

The biggest differences: blocks are marked with curly braces instead of indentation (as in C), statements end with semicolons, and you should compare with `===`, not `==`. (JavaScript's `==` tries to convert types first, so `"5" == 5` is true. `===` does not.)

Every browser has a JavaScript console built in. Open it with F12 (or Cmd-Option-J on a Mac in Chrome), type `2 + 3`, and press Enter. It is a great place to try things out.

6.7 Making the Page Respond

When the browser reads your HTML, it builds a tree of objects, one for every element. This tree is called the *Document Object Model* (DOM). JavaScript can find any element in the tree, change it, and tell the browser to run a function when something happens to it, such as a click. That function is called an *event listener*.

Create `script.js` in the public folder, and load it by adding one more line to the `<head>`:

```
1 <script src="script.js" defer></script>
```

The word `defer` tells the browser to wait until it has built the whole page before running the script. Without it, your script would run before the elements it is looking for exist.

6.7.1 Dark Mode

Add a button inside the `<header>`, just below the image:

```
1 <div class="controls">
```

```
2 <button id="theme-button">Dark mode</button>
3 </div>
```

Add the dark colors to `style.css`:

```
1 body.dark {
2   background-color: #1e1e1e;
3   color: #eeeeee;
4 }
5
6 body.dark h1,
7 body.dark h2 {
8   color: #f4a460;
9 }
```

The selector `body.dark` means “the body, but only when it has the class `dark`.” Right now, it never does. JavaScript will add the class. Put this in `script.js`:

```
1 // Dark mode
2 const themeButton = document.querySelector("#theme-button");
3
4 themeButton.addEventListener("click", () => {
5   document.body.classList.toggle("dark");
6   if (document.body.classList.contains("dark")) {
7     themeButton.textContent = "Light mode";
8   } else {
9     themeButton.textContent = "Dark mode";
10  }
11 });
```

Here is what each piece does:

- `document.querySelector` takes a CSS selector and returns the first element that matches.
- `addEventListener("click", ...)` says “when this button is clicked, run this function.”
- `() => { ... }` is a function with no name and no parameters, written on the spot. JavaScript programmers call it an *arrow function*. It is like Python’s `lambda`, but its body can hold many lines.
- `classList.toggle("dark")` adds the class if it is missing and removes it if it is there.
- `textContent` is the text inside an element. Changing it changes the page.

Notice the division of labor. JavaScript does not set any colors. It only flips a class, and CSS decides what that class looks like. If you later want a different dark theme, you will

not touch the JavaScript.

6.7.2 Crossing Off Steps

When you are baking with buttery fingers, it is easy to lose your place. Let a click on a step cross it off. In `style.css`:

```
1 .steps-section li {  
2   cursor: pointer;  
3 }  
4  
5 .done {  
6   text-decoration: line-through;  
7   color: #999999;  
8 }
```

In `script.js`:

```
1 // Click a step to cross it off  
2 for (const step of document.querySelectorAll(".steps-section li")) {  
3   step.addEventListener("click", () => {  
4     step.classList.toggle("done");  
5   });  
6 }
```

`querySelectorAll` returns *every* matching element, and the loop gives each step its own listener. The `cursor: pointer` rule turns the mouse pointer into a hand over the steps, a hint that they can be clicked.

6.7.3 Doubling the Recipe

Now for the most useful feature: a box where you type the number of batches, and every amount on the page updates. JavaScript needs to know the original amount of each ingredient, so store it in the HTML using a *data attribute*. Any attribute whose name starts with `data-` is yours to invent. Change the ingredient list to wrap each amount in a `<span>`:

```
1 <ul>  
2   <li><span class="amount" data-amount="2.25">2 1/4</span> cups flour</li>  
3   <li><span class="amount" data-amount="1">1</span> teaspoon baking soda</li>  
4   <li><span class="amount" data-amount="1">1</span> teaspoon salt</li>  
5   <li><span class="amount" data-amount="1">1</span> cup butter</li>  
6 </ul>
```

```

6 <li><span class="amount" data-amount="0.75">3/4</span> cup sugar</li>
7 <li><span class="amount" data-amount="0.75">3/4</span> cup brown sugar</li>
8 <li><span class="amount" data-amount="2">2</span> eggs</li>
9 <li><span class="amount" data-amount="2">2</span> cups chocolate chips</li>
10 </ul>

```

Do the same for the number of cookies in the Recipe Facts table:

```

1 <td><span class="amount" data-amount="24">24</span> cookies</td>

```

A `<span>` is like a `<div>` that sits inside a line of text instead of starting a new block.

Then add a number box at the start of the controls:

```

1 <label>Batches: <input type="number" id="batches" value="1" min="1"
  ↪ step="1"></label>

```

Bakers measure in quarter cups, not decimals, so the script needs a function that writes 2.25 as “2 1/4”. It separates the whole part from the fraction and counts the quarters:

```

1 // Scale the recipe
2 function formatAmount(x) {
3   const whole = Math.floor(x);
4   const quarters = Math.round((x - whole) * 4);
5   const fraction = ["", "1/4", "1/2", "3/4"][quarters];
6   if (whole === 0) {
7     return fraction;
8   }
9   if (fraction === "") {
10    return String(whole);
11  }
12  return whole + " " + fraction;
13 }
14
15 const batchesInput = document.querySelector("#batches");
16
17 batchesInput.addEventListener("input", () => {
18   const batches = Number(batchesInput.value);
19   if (batches < 1) {
20     return;
21   }
22   for (const span of document.querySelectorAll(".amount")) {
23     span.textContent = formatAmount(Number(span.dataset.amount) * batches);
24   }
25 });

```

The input event fires every time the value in the box changes. `span.dataset.amount` reads the `data-amount` attribute, which is always a string, so `Number` converts it, just as you would use `float` in Python. If the box is empty, `Number("")` is 0, and the return skips the update rather than erasing every amount.

Example: What does the page show for flour with 3 batches?

Solution: $2.25 \times 3 = 6.75$. The whole part is 6, and $0.75 \times 4 = 3$ quarters, so the fraction is "3/4". The page shows "6 3/4 cups flour".

Finally, tidy up the controls so they sit in a neat row:

```

1  .controls {
2      display: flex;
3      flex-wrap: wrap;
4      align-items: center;
5      gap: 0.6rem;
6  }
7
8  button,
9  input {
10     font: inherit;
11 }
12
13 button {
14     background-color: white;
15     border: 1px solid #7b3f00;
16     border-radius: 8px;
17     padding: 0.2rem 0.8rem;
18     cursor: pointer;
19 }
20
21 #batches {
22     width: 4rem;
23 }
```

(`font: inherit` makes buttons and boxes use the page's font; browsers give them a smaller one by default.)

6.8 Talking to Your Server

Everything so far happens inside one person's browser. If you click "Dark mode," your friend's screen does not change. To share something between visitors, such as a count of likes, the page has to talk to a server.

6.8.1 Serving the Page with Express

In the Building a Web API chapter, you built an Express server that answered requests with JSON. The same server can also send your HTML, CSS, JavaScript, and images. Create `server.js` in the recipe folder (*not* inside `public`):

```
1  const express = require("express");
2
3  const app = express();
4  const PORT = 3000;
5
6  let likes = 0;
7
8  // Send the files in the public folder: index.html, style.css, script.js, ...
9  app.use(express.static("public"));
10
11 app.get("/api/likes", (req, res) => {
12   res.json({ likes: likes });
13 });
14
15 app.post("/api/likes", (req, res) => {
16   likes += 1;
17   res.json({ likes: likes });
18 });
19
20 app.listen(PORT, () => {
21   console.log(`Recipe site running at http://localhost:${PORT}`);
22 });
```

`express.static("public")` tells Express to answer any request for a file in `public` with that file, and to send `index.html` when someone asks for the site itself. The two other routes are an API with a single number in it: GET reads the likes, and POST adds one.

Start it from the recipe folder, as you did in the Building a Web API chapter:

```
1  npm init -y
2  npm install express
3  node server.js
```

Now visit <http://localhost:3000> in your browser. Your page should look exactly as before, but now it is coming from your server instead of a file.

6.8.2 The Like Button

Add the button to the controls in `index.html`:

```
1 <button id="like-button">&#10084; <span id="like-count">0</span></button>
```

(❤ is the HTML code for a heart.) And add this to `script.js`:

```
1 // Likes, stored on the server
2 const likeButton = document.querySelector("#like-button");
3 const likeCount = document.querySelector("#like-count");
4
5 async function updateLikes(method) {
6   const response = await fetch("/api/likes", { method: method });
7   const data = await response.json();
8   likeCount.textContent = data.likes;
9 }
10
11 updateLikes("GET");
12 likeButton.addEventListener("click", () => {
13   updateLikes("POST");
14 });
```

`fetch` sends an HTTP request, just as `curl` did in the Building a Web API chapter. The network is slow compared to the computer, so `fetch` does not make the page freeze while it waits. Instead, it returns a *promise*: an object that will hold the response once it arrives. Inside a function marked `async`, the word `await` pauses that function (and only that function) until the promise is ready. Meanwhile, the rest of the page keeps responding to clicks.

When the page loads, `updateLikes("GET")` asks the server for the current count. Each click sends a `POST`, and the server replies with the new total. Open the page in two browser windows, click in one, and reload the other: the count is shared.

The like button only works when the page comes from the server. If you open `index.html` by double-clicking it, there is no server to answer `/api/likes`, and the console will show an error.

6.9 Showing Your Friends

Your server is running on your computer, so `localhost` only works for you. To let a friend on the same Wi-Fi network visit, find your computer's local IP address (on a Mac, run `ipconfig getifaddr en0` in the terminal; on Windows, run `ipconfig` and look for the IPv4

address). If it is 192.168.1.23, your friend can visit `http://192.168.1.23:3000` on their phone. Your computer's firewall may ask whether to allow the connection.

To put the site on the internet for anyone to visit, you would use a hosting service. Several services host Node apps, often free for small projects. If you drop the like button, the site needs no server code at all, and free static hosts such as GitHub Pages will serve it.

One warning: everything in `index.html`, `style.css`, and `script.js` is sent to every visitor, who can read it with "View Page Source." Never put a password or an API key, like the ones in the Building a Web API chapter, in your front-end code. Secrets belong on the server.

Exercise 34 Which Color Wins?

A stylesheet has these three rules:

Working Space

```
1 p { color: red; }
2 .intro { color: green; }
3 #recipe-title { color: blue; }
```

What color is each of these elements?

```
1 <p class="intro">
2 <h1 id="recipe-title" class="intro">
```

Answer on Page 127

Exercise 35 How Much Room?

An element has this style:

Working Space

```
1 width: 300px;
2 padding: 20px;
3 border: 2px solid black;
4 margin: 10px;
```

How wide is the visible box, from the outside of the left border to the outside of the right border? How much horizon-

tal space does it take up on the page?

Answer on Page 127

Exercise 36 Highlight the Ingredient

Write a CSS rule that gives an ingredient a light tan background (#fdebd0) while the mouse is over it.

Working Space

Answer on Page 127

Exercise 37 Python to JavaScript

Translate this Python into JavaScript, and run it in your browser's console:

```
1 def fahrenheit_to_celsius(f):  
2     return (f - 32) * 5 / 9  
3  
4 for temp in [350, 375, 400]:  
5     print(temp, "F is",  
        ↪ round(fahrenheit_to_celsius(temp)),  
        ↪ "C")
```

Working Space

Answer on Page 127

Exercise 38 2 Cup Butter?

With 2 batches, the page says “2 cup butter” and “1 1/2 cup sugar.” Fix it.

Hint: move the unit into the span, and

Working Space

store it in a second data attribute:

```
1 <span class="amount" data-amount="1"  
2 data-unit="cup">1 cup</span> butter
```

Then have the script add an “s” when the amount is more than 1.

Answer on Page 128

Exercise 39 Progress Counter

Add this paragraph below the steps:

Working Space

```
1 <p id="progress">0 of 7 steps done</p>
```

Change the step-clicking code so that the paragraph always shows how many steps are crossed off.

Answer on Page 128

Exercise 40 Where Did the Likes Go?

Stop the server by pressing Ctrl-C, and start it again. What happens to the likes? Why? Change the server so that the count survives a restart.

Hint: Node’s built-in `fs` module has functions to read and write files.

Working Space

Answer on Page 129

Exercise 41 Cheating the Count

A friend runs this command in a loop
1,000 times:

```
1 curl -X POST  
  ↪ http://192.168.1.23:3000/api/likes
```

What happens? Why can the server not simply ignore requests that do not come from the like button?

Working Space

Answer on Page 129

Digital Logic: From Transistors to Adders

Type $6 + 7$ into your phone's calculator and it answers 13 before your finger leaves the screen. Inside the phone, there is no one who knows arithmetic. There are only billions of tiny switches, each of which is either on or off. So how does a pile of switches add?

In this chapter, you will build an adding machine from scratch. You will start with a single transistor, make a gate called NAND out of four of them, build every other kind of logic from NAND alone, and finally chain the pieces into a circuit that adds binary numbers. Then you will check your design in Python.

You already have the pieces. The Sets and Logic chapter gave you logic tables. The Diodes chapter showed how doping turns silicon into a conductor. The Data Representation chapter showed how numbers are stored in binary, including negative numbers in two's complement.

7.1 Bits as Voltages

A wire in a chip carries a bit as a voltage. A wire near 0 volts means 0 (false). A wire near the chip's supply voltage, called V_{dd} and often about 1 volt in a modern phone, means 1 (true). In between is a no-man's-land that a well-designed circuit passes through quickly.

From here on, we write logic tables with 1 and 0 instead of T and F. The tables in this form are usually called *truth tables*.

7.2 Transistors Are Switches

A *transistor* is a switch with no moving parts, controlled by a voltage. The kind found in nearly every chip is the MOSFET. It is a strip of doped silicon with a third terminal, the *gate*, sitting just above it behind a thin layer of glass. Putting a voltage on the gate pulls charge carriers into the silicon under it, opening a path for current. Remove the voltage, and the path closes.

There are two flavors, and they behave as opposites:

- An **nMOS** transistor is *on* (conducting) when its gate is 1.
- A **pMOS** transistor is *on* when its gate is 0. Its symbol has a little circle on the gate, which means “opposite.”

Chips pair the two flavors together, a style called *CMOS* (complementary MOS). Here is the simplest CMOS circuit, the *inverter*:

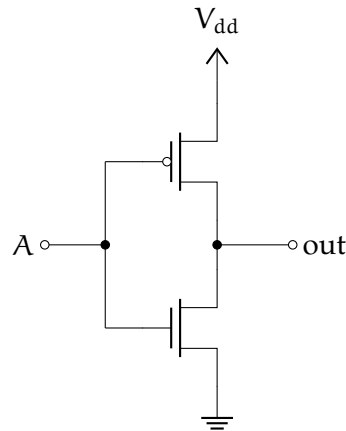


Figure 7.1: A CMOS inverter. The pMOS transistor on top connects the output to V_{dd} ; the nMOS transistor on the bottom connects it to ground (0 V).

Trace it. If $A = 1$, the bottom transistor is on and the top one is off, so the output is connected to ground: $\text{out} = 0$. If $A = 0$, the top transistor is on and the bottom is off, so the output is connected to V_{dd} : $\text{out} = 1$. The circuit flips its input. It is a NOT gate.

Notice that one transistor is always off, so no current flows straight from V_{dd} to ground. A CMOS circuit uses power mostly when its outputs switch. This is why your phone can hold billions of transistors without melting.

7.3 The NAND Gate

Now use four transistors. Put two pMOS transistors side by side (in parallel) between V_{dd} and the output, and two nMOS transistors one after the other (in series) between the output and ground.

The output reaches ground only if *both* bottom transistors are on, which needs $A = 1$ and $B = 1$. In every other case, at least one of the top transistors is on and pulls the output up to 1.

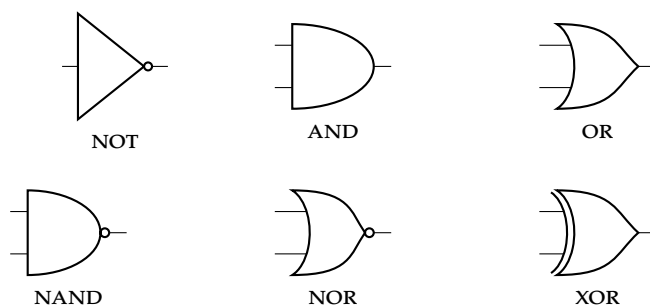


Figure 7.3: The common logic gates.

7.5 NAND Can Build Anything

Here is the surprise: you never need any gate except NAND. Every other gate can be built from NANDs.

NOT: Feed the same signal into both inputs. $\text{NAND}(A, A)$ is 0 when $A = 1$ and 1 when $A = 0$.

AND: A NAND followed by a NOT. That is 2 NAND gates.

OR: Recall the exercise in the Sets and Logic chapter, where you showed that $\neg(A \text{ or } B)$ is the same as $(\neg A)$ and $(\neg B)$. This is one of *De Morgan's laws*. Flip both sides and you get the form we need:

$$A \text{ or } B = \text{NAND}(\neg A, \neg B)$$

So OR is a NAND whose inputs are first inverted: 3 NAND gates.

XOR: This one takes 4 NAND gates, wired like this:

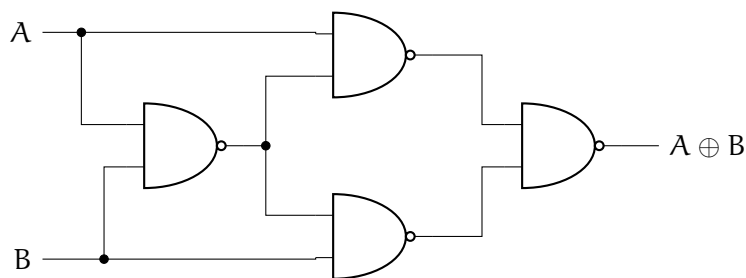


Figure 7.4: XOR built from four NAND gates.

Example: Check the XOR circuit when $A = 1$ and $B = 0$.

Solution: The first gate gives $\text{NAND}(1, 0) = 1$. The top gate gives $\text{NAND}(A, 1) = \text{NAND}(1, 1) = 0$. The bottom gate gives $\text{NAND}(B, 1) = \text{NAND}(0, 1) = 1$. The last gate gives $\text{NAND}(0, 1) = 1$. And indeed $1 \oplus 0 = 1$.

A gate that can build every other gate is called *universal*. This is why chip designers sometimes say that a computer is just NAND gates all the way down.

7.6 Adding in Binary

You add binary numbers the way you learned to add decimal numbers in elementary school: column by column, from right to left, carrying when a column overflows. The only difference is that in binary, $1 + 1 = 10_2$, so you write 0 and carry 1.

Example: Add 0110_2 (which is 6) and 0111_2 (which is 7).

Solution:

$$\begin{array}{rcccc}
 & 1 & 1 & 0 & \text{(carries)} \\
 & 0 & 1 & 1 & 0 \\
 + & 0 & 1 & 1 & 1 \\
 \hline
 & 1 & 1 & 0 & 1
 \end{array}$$

Right column: $0 + 1 = 1$. Next: $1 + 1 = 10_2$, write 0, carry 1. Next: $1 + 1 + 1 = 11_2$, write 1, carry 1. Last: $0 + 0 + 1 = 1$. The answer is $1101_2 = 8 + 4 + 1 = 13$.

7.6.1 The Half Adder

Look at the rightmost column, where there is no carry coming in. It adds two bits, A and B, and produces a sum bit and a carry bit:

A	B	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Compare with the gate table. The sum column is exactly XOR, and the carry column is exactly AND. A circuit with one of each is called a *half adder*.

7.6.2 The Full Adder

Every other column has three bits to add: A, B, and the carry coming in from the right. A *full adder* handles this with two half adders. The first adds A and B. The second adds

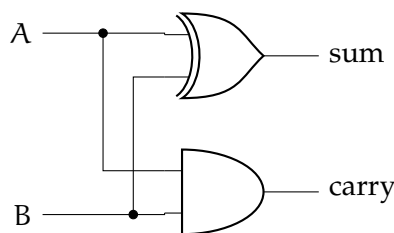


Figure 7.5: A half adder: XOR gives the sum bit; AND gives the carry.

that sum to the incoming carry. If either half adder produces a carry, there is a carry out, so their two carries go into an OR gate.

$$\begin{aligned}\text{sum} &= (A \oplus B) \oplus C_{\text{in}} \\ C_{\text{out}} &= (A \text{ and } B) \text{ or } ((A \oplus B) \text{ and } C_{\text{in}})\end{aligned}$$

Built carefully, a full adder takes just 9 NAND gates, which is 36 transistors.

7.6.3 The Ripple-Carry Adder

To add two 4-bit numbers, line up four full adders, one per column. Each carry out becomes the carry in of the next column to the left. The carry into the rightmost column is 0.

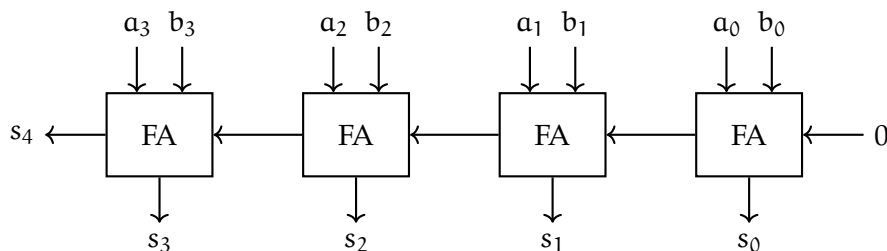


Figure 7.6: A 4-bit ripple-carry adder. Bit 0 is the least significant bit, on the right. The final carry out becomes the fifth bit of the answer.

The carry “ripples” from right to left, just as it does when you add with a pencil. A 64-bit adder is simply 64 full adders in a row.

7.7 Checking the Design in Python

Before a company spends millions of dollars making a chip, engineers simulate every circuit. Let us do the same. The program below defines a single gate, nand, and builds

everything else from it. It even counts how many NAND gates it uses. Create a file called `adder.py`:

```
1 nand_count = 0
2
3 def nand(a, b):
4     global nand_count
5     nand_count += 1
6     return 0 if (a == 1 and b == 1) else 1
7
8 # Every other gate is built from NAND
9 def not_(a):
10     return nand(a, a)
11
12 def and_(a, b):
13     return not_(nand(a, b))
14
15 def or_(a, b):
16     return nand(not_(a), not_(b))
17
18 def xor(a, b):
19     n = nand(a, b)
20     return nand(nand(a, n), nand(b, n))
21
22 def full_adder(a, b, carry_in):
23     s = xor(a, b)
24     total = xor(s, carry_in)
25     carry_out = or_(and_(a, b), and_(s, carry_in))
26     return total, carry_out
27
28 def add(a_bits, b_bits):
29     # Bits are listed from least significant to most significant
30     carry = 0
31     result = []
32     for a, b in zip(a_bits, b_bits):
33         total, carry = full_adder(a, b, carry)
34         result.append(total)
35     result.append(carry)
36     return result
37
38 def to_bits(n, width):
39     return [(n >> i) & 1 for i in range(width)]
40
41 def to_int(bits):
42     return sum(bit << i for i, bit in enumerate(bits))
43
44 print("6 + 7 =", to_int(add(to_bits(6, 4), to_bits(7, 4))))
45
46 # Test every pair of 4-bit numbers
47 errors = 0
48 for x in range(16):
49     for y in range(16):
```

```

50         if to_int(add(to_bits(x, 4), to_bits(y, 4))) != x + y:
51             errors += 1
52     print("pairs tested: 256, errors:", errors)
53
54     nand_count = 0
55     add(to_bits(6, 4), to_bits(7, 4))
56     print("NAND gates used for one 4-bit addition:", nand_count)

```

The names `not_`, `and_`, and `or_` end with an underscore because `not`, `and`, and `or` are already Python keywords. The function `to_bits` uses `>>` (shift right) and `&` (bitwise AND) to pull out one bit at a time.

When you run it, you should see:

```

6 + 7 = 13
pairs tested: 256, errors: 0
NAND gates used for one 4-bit addition: 60

```

Every one of the 256 possible additions is correct. Our straightforward full adder uses $4 + 4 + 2 + 2 + 3 = 15$ NAND gates, so 4 bits take 60. The clever 9-gate version is an exercise.

7.8 Subtraction for Free

In the Data Representation chapter, you learned that in two's complement, you negate a number by flipping every bit and adding 1. So

$$A - B = A + (\text{flipped } B) + 1$$

A chip does not need a separate subtracting circuit. It runs `B` through a row of inverters and sets the carry into the rightmost full adder to 1 instead of 0. That “+1” comes for free.

7.9 From Adders to Computers

Your adder is the core of the *arithmetic logic unit* (ALU), the part of a processor that does the math. A few more ideas turn it into a computer:

- **Memory.** Wire two NAND gates in a loop, each feeding the other, and the pair can hold a bit after its inputs go away. Thousands of these store the numbers being added.
- **A clock.** A signal that ticks billions of times per second tells every part when to take its next step.

- **Control.** Instructions, stored as bits like everything else, choose which operation the ALU performs on each tick.

A modern phone chip holds tens of billions of transistors. Each one is just a switch, but arranged the right way, they add, compare, store, and decide. That is how a computer actually computes.

Exercise 42 NOR Truth Table

Make a truth table for $\neg(A \text{ or } B)$. Which gate is this?

Working Space

Answer on Page 130

Exercise 43 NOR Is Universal Too

Show how to build NOT and OR using only NOR gates.

Working Space

Answer on Page 130

Exercise 44 Add by Hand

Add 1011_2 and 0110_2 in binary. Check your answer in decimal. Why does the answer need 5 bits?

Working Space

Answer on Page 130

Exercise 45 Trace a Full Adder

Find the sum and carry out of a full adder when $A = 1$, $B = 1$, and $C_{in} = 1$, using the two formulas for a full adder.

Working Space

Answer on Page 130

Exercise 46 Counting Transistors

Using 9 NAND gates per full adder and 4 transistors per NAND gate, how many transistors does a 64-bit ripple-carry adder need?

Working Space

Answer on Page 130

Exercise 47 The Clever Full Adder

Rewrite `full_adder` to use only 9 NAND gates. Check that the test still finds 0 errors and that the count drops to 36.

Hint: build each XOR from 4 NANDs, and save the output of the first gate of each:

$$n_1 = \text{NAND}(A, B)$$

$$m_1 = \text{NAND}(A \oplus B, C_{in})$$

Then $C_{out} = \text{NAND}(n_1, m_1)$.

Working Space

Answer on Page 130

Exercise 48 **Waiting for the Carry**

Each gate takes a tiny amount of time to switch. Why is a ripple-carry adder slow for 64-bit numbers?

Working Space

Answer on Page 131

Exercise 49 **Subtract with an Adder**

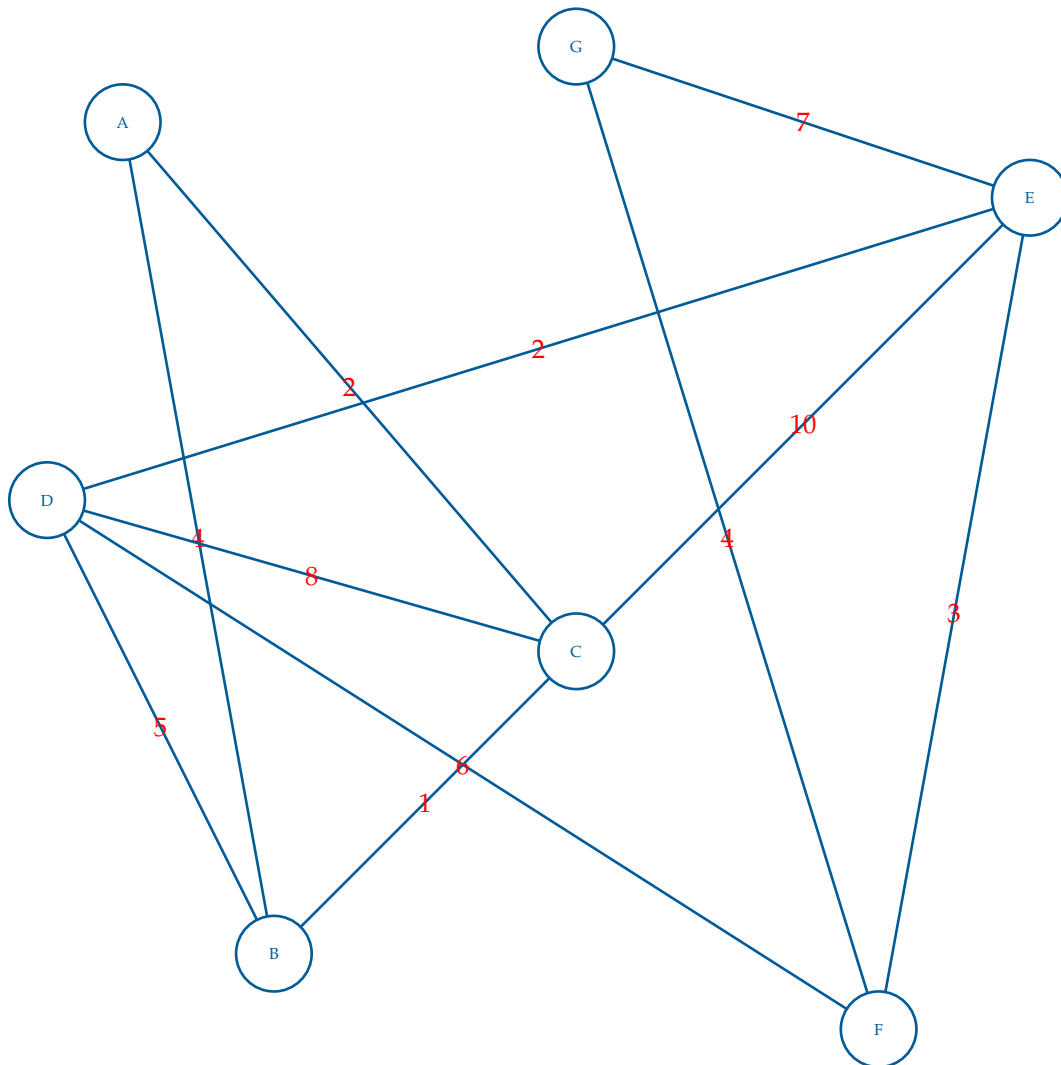
Write a function `subtract` that takes two lists of bits, flips every bit of the second list with `not_`, starts the carry at 1, and adds. Use it to compute $13 - 6$ with 4 bits.

Working Space

Answer on Page 131

Answers to Exercises

Answer to Exercise 1 (on page 7)



Answer to Exercise 2 (on page 8)

Add G-F (4), then F-E (3), then E-D (2), then D-B (5), then B-C (1), then C-A (2). The

total weight is $4 + 3 + 2 + 5 + 1 + 2 = 17$, which matches the worked example.

Answer to Exercise 3 (on page 11)

1. Injective, but not surjective. No two inputs map to the same output, but d is never reached.
2. Surjective, but not injective. Every element of the codomain is reached, but both 1 and 4 map to a .
3. Bijective. Every element of the codomain is reached exactly once.

Answer to Exercise 4 (on page 18)

Assuming FindOdd exists, we can make the following pseudocode.

IsBipartite(G)

Require: G , a graph on which DFS has been run

Ensure: **true** if G is bipartite, **false** otherwise

- 1: **for** all back edges e **do**
- 2: **if** FindOdd(G, e) = **true** **then**
- 3: **return false**
- 4: **end if**
- 5: **end for**
- 6: **return true**

Answer to Exercise 5 (on page 19)

Remove the edge $e = (u, v)$ from G .

Run BFS or DFS starting from u .

if v is reached **then**

 Return TRUE.

else

 Return FALSE.

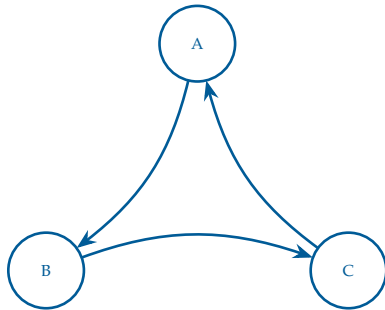
end if

Answer to Exercise 6 (on page 19)

Proof. Build a DFS tree T of G . T spans G , so if G has at least two vertices, T has at least

two leaves. Let l be a leaf of T . Removing a leaf from a tree leaves a tree, so $T - l$ is connected. Every vertex other than l is in $T - l$, and $T - l$ uses only edges of $G - l$. So $G - l$ is connected, and removing l is the vertex we need. $\square$

Answer to Exercise 7 (on page 20)



A directed triangle $A \rightarrow B \rightarrow C \rightarrow A$. You can get from any vertex to any other by going around the cycle, so it is strongly connected. Remove any one vertex, say A , and only the edge $B \rightarrow C$ is left: you can reach C from B but not B from C .

Answer to Exercise 8 (on page 29)

$5,000,000/1,460 \approx 3,424.7$, so it takes 3,425 packets, with the last one only partly full.

Answer to Exercise 9 (on page 30)

TCP would stop to resend lost packets, and the call would freeze while it waited. By the time a resent packet arrived, the moment it belongs to would already be over. With UDP, a lost packet causes a tiny glitch, and the call keeps going in real time.

Answer to Exercise 10 (on page 30)

An IPv4 address has 32 bits. In a /16 block, the first 16 bits are fixed and the last 16 are free, so there are $2^{16} = 65,536$ addresses. In a /8 block, 24 bits are free: $2^{24} = 16,777,216$ addresses. You can check the first with `ipaddress.ip_network("10.0.0.0/16").num_addresses`.

Answer to Exercise 11 (on page 30)

$2^{128}/(8.1 \times 10^9) \approx 3.4 \times 10^{38}/8.1 \times 10^9 \approx 4.2 \times 10^{28}$ addresses per person. IPv4 cannot even give each person one.

Answer to Exercise 12 (on page 30)

From D: F costs 2 (next hop F), E costs 6 (next hop E), C costs 10 (next hop E), B costs 13 (next hop B), and A costs 17 (next hop E, through E and C). The path to B through E and C also costs 13; the program keeps the direct link because it found it first. To check, add `show(forwarding_table(links, "D"))` before the lines that remove the link.

Answer to Exercise 13 (on page 31)

The round trip is $2 \times 9,330 = 18,660$ km, and $18,660/200,000 \approx 0.093$ seconds, or about 93 ms, before any router delays. In a fast game, a tenth of a second between pressing a button and seeing the result is easy to feel. Game companies put servers near their players for this reason.

Answer to Exercise 14 (on page 31)

A long TTL means resolvers rarely need to ask again, so lookups are fast and the site's DNS servers get few requests. But after the move, resolvers may keep handing out the old, cached address for up to a week, and some visitors will not find the site. Sites usually shorten the TTL a few days before a planned move.

Answer to Exercise 15 (on page 31)

Answers vary, but most routes take somewhere between 8 and 20 hops. The first hop is your home router. A sudden jump of tens of milliseconds usually means the packet crossed a long distance, such as across a country or under an ocean. Some routers do not reply at all and show up as asterisks.

Answer to Exercise 16 (on page 35)

```
2 <html>
3 <head>
4   <title>Chocolate Chip Cookies</title>
5 </head>
6 <body>
7   <h1 id="recipe-title">Chocolate Chip Cookies</h1>
8   <p class="intro" title="A short recipe description">
9     This recipe makes soft and sweet chocolate chip cookies.
10  </p>
11 </body>
12 </html>
```

Answer to Exercise 17 (on page 38)

```
1 <body>
2   <div class="recipe-header">
3     <h1 id="recipe-title">Chocolate Chip Cookies</h1>
4     <p class="intro" title="A short recipe description">
5       This recipe makes soft and sweet chocolate chip cookies.
6     </p>
7   </div>
8
9   <div class="ingredients-section">
10    <p>Ingredients</p>
11  </div>
12
13  <div class="steps-section">
14    <p>Steps</p>
15  </div>
16 </body>
```

Answer to Exercise 18 (on page 41)

```
1 <body>
2   <header>
3     <h1 id="recipe-title">Chocolate Chip Cookies</h1>
4     <p class="intro" title="A short recipe description">
5       This recipe makes soft and sweet chocolate chip cookies.
6     </p>
7   </header>
8
9   <main>
10    <section class="ingredients-section">
11      <p>Ingredients</p>
12    </section>
```

```
13     <section class="steps-section">
14         <p>Steps</p>
15     </section>
16 </main>
17
18 <footer>
19 </footer>
20 </body>
```

Answer to Exercise 19 (on page 42)

```
1 <main>
2   <section class="ingredients-section">
3     <h2 id="ingredients">Ingredients</h2>
4   </section>
5
6   <section class="steps-section">
7     <h2 id="steps">Steps</h2>
8   </section>
9 </main>
```

Answer to Exercise 20 (on page 44)

```
1 <footer>
2   <p><strong>Important Note:</strong> Make sure not to overbake the cookies, or
3     ↳ they will become hard.</p>
4 </footer>
```

Answer to Exercise 21 (on page 47)

```
1 <section class="ingredients-section">
2   <h2 id="ingredients">Ingredients</h2>
3   <ul>
4     <li>2 1/4 cups flour</li>
5     <li>1 teaspoon baking soda</li>
6     <li>1 teaspoon salt</li>
7     <li>1 cup butter</li>
8     <li>3/4 cup sugar</li>
9     <li>3/4 cup brown sugar</li>
```

```

10     <li>2 eggs</li>
11     <li>2 cups chocolate chips</li>
12 </ul>
13 </section>
14
15 <section class="steps-section">
16   <h2 id="steps">Steps</h2>
17   <ol>
18     <li>Preheat the oven to 375°F.</li>
19     <li>Mix the dry ingredients in one bowl.</li>
20     <li>Mix the butter, sugar, brown sugar, and eggs in another bowl.</li>
21     <li>Stir the dry ingredients into the wet ingredients.</li>
22     <li>Add the chocolate chips.</li>
23     <li>Scoop the dough onto a baking sheet.</li>
24     <li>Bake for 9 to 11 minutes.</li>
25   </ol>
26 </section>

```

Answer to Exercise 22 (on page 49)

```

1 <header>
2   <h1 id="recipe-title">Chocolate Chip Cookies</h1>
3   <p class="intro" title="A short recipe description">
4     This recipe makes soft and sweet chocolate chip cookies.
5   </p>
6
7   <nav>
8     <ul>
9       <li><a href="#ingredients">Ingredients</a></li>
10      <li><a href="#steps">Steps</a></li>
11    </ul>
12  </nav>
13 </header>

```

Answer to Exercise 23 (on page 52)

```

1 <section>
2   <h2>Recipe Facts</h2>
3   <table border="1">
4     <caption>Chocolate Chip Cookie Recipe Facts</caption>
5     <tbody>
6       <tr>
7         <td>Prep Time</td>
8         <td>15 minutes</td>
9       </tr>

```

```

10     <tr>
11         <td>Bake Time</td>
12         <td>10 minutes</td>
13     </tr>
14     <tr>
15         <td>Servings</td>
16         <td>24 cookies</td>
17     </tr>
18 </tbody>
19 </table>
20 </section>

```

Answer to Exercise 24 (on page 57)

```

1 <header>
2   <h1 id="recipe-title">Chocolate Chip Cookies</h1>
3   <p class="intro" title="A short recipe description">
4     This recipe makes soft and sweet chocolate chip cookies.
5   </p>
6
7   <nav>
8     <ul>
9       <li><a href="#ingredients">Ingredients</a></li>
10      <li><a href="#steps">Steps</a></li>
11    </ul>
12  </nav>
13
14  
16 </header>

```

Answer to Exercise 25 (on page 68)

1. GET, because it only reads data.
2. POST, because it creates something new.
3. PATCH, because it changes one field and leaves the rest alone. (Some APIs use PUT and send the whole profile.)
4. DELETE.

Answer to Exercise 26 (on page 68)

`data["current"]["temperature_2m"]`. It is a float (59.4), because JSON numbers with a decimal point become Python floats.

Answer to Exercise 27 (on page 68)

Change the two numbers in the `params` dictionary. For example, Chicago is at about latitude 41.88 and longitude -87.63 . Longitudes west of Greenwich, England, are negative, and latitudes south of the equator are negative.

Answer to Exercise 28 (on page 69)

`data["daily"]` holds three lists of the same length: the dates, the highs, and the lows. Python's `zip` walks through them together:

```
1 daily = data["daily"]
2 for day, high, low in zip(daily["time"], daily["temperature_2m_max"],
3                           daily["temperature_2m_min"]):
4     print(day, "high", high, "low", low)
```

It prints something like `2026-10-22 high 77.7 low 55.9`, one line per day.

Answer to Exercise 29 (on page 69)

The status code is 400, and the JSON says `{'error': True, 'reason': 'Latitude must be in range of -90 to 90°. Given: 139.74.'}`. (The program then crashes, because there is no `"current"` key in an error response. Real programs check `response.status_code` before using the data.) It is a 400 because the request itself was bad: no place on Earth has a latitude of 139.74. A 404 would mean the address you asked for does not exist on the server, but the `/v1/forecast` endpoint is there.

Answer to Exercise 30 (on page 82)

```
1 curl -X PATCH http://localhost:3000/students/43 \
2     -H "Content-Type: application/json" \
```

```
3 -d '{ "credits_completed": 70 }'
```

The response is the updated student, {"student_id":43,"name":"Jane Doe","major":"Physics","credits_completed":70}. PATCH changes only the field that was sent, so the name and major stay the same.

Answer to Exercise 31 (on page 84)

1. 200: John Smith is found.
2. 404: there is no student 19.
3. 201: a new student was created.
4. 204 the first time (deleted, nothing to send back), then 404 (already gone).
5. 401: the client has not proven who it is.

You can check each one with `curl -i`, which shows the status line.

Answer to Exercise 32 (on page 84)

```
1 app.get("/students", (req, res) => {
2   const major = req.query.major;
3   if (major) {
4     return res.json(students.filter((s) => s.major === major));
5   }
6   res.json(students);
7 });
```

Then `curl "http://localhost:3000/students?major=Mathematics"` returns only Jane Doe. (Put the URL in quotes, or the shell may treat the `?` as a special character.)

Answer to Exercise 33 (on page 84)

```
1 app.post("/students", (req, res) => {
2   const student = req.body || {};
3   if (typeof student.name !== "string" || student.name === "") {
4     return res.status(400).json({ error: "name must be a non-empty string" });
5   }
6 }
```

```

6   if (typeof student.credits_completed !== "number" || student.credits_completed <
    ↪  0) {
7     return res.status(400).json({ error: "credits_completed must be a number 0 or
    ↪  more" });
8   }
9   students.push(student);
10  res.status(201).json(student);
11  });

```

typeof returns the type of a value as a string, such as "string" or "number". The || {} handles a request with no body at all. A POST with "credits_completed": -5 now gets {"error": "credits_completed must be a number 0 or more"} and a 400.

Answer to Exercise 34 (on page 100)

The paragraph is green: a class selector is more specific than an element selector. The heading is blue: it matches both .intro and #recipe-title, and an id beats a class.

Answer to Exercise 35 (on page 100)

By default, width sets only the content. The visible box is $300 + 2(20) + 2(2) = 344$ pixels wide. With the margins, it takes up $344 + 2(10) = 364$ pixels. (Many designers add box-sizing: border-box, which makes width include the padding and border, so the visible box is exactly 300 pixels.)

Answer to Exercise 36 (on page 101)

```

1  .ingredients-section li:hover {
2    background-color: #fdebd0;
3  }

```

Answer to Exercise 37 (on page 101)

```

1  function fahrenheitToCelsius(f) {
2    return (f - 32) * 5 / 9;
3  }
4
5  for (const temp of [350, 375, 400]) {

```

```

6 console.log(temp, "F is", Math.round(fahrenheitToCelsius(temp)), "C");
7 }

```

It prints:

```

350 F is 177 C
375 F is 191 C
400 F is 204 C

```

JavaScript programmers usually name things in camelCase rather than with underscores.

Answer to Exercise 38 (on page 101)

Give every ingredient that has a unit a data-unit attribute holding the singular unit (cup or teaspoon), and move the unit inside the span, as in the hint. Leave the eggs and cookies without one. Then replace the loop in the input listener:

```

1 for (const span of document.querySelectorAll(".amount")) {
2   const amount = Number(span.dataset.amount) * batches;
3   let text = formatAmount(amount);
4   if (span.dataset.unit) {
5     text += " " + span.dataset.unit;
6     if (amount > 1) {
7       text += "s";
8     }
9   }
10  span.textContent = text;
11 }

```

If an element has no data-unit attribute, span.dataset.unit is undefined, which counts as false, so eggs and cookies get no unit. Now 3/4 cup stays singular, and 1 1/2 cups becomes plural.

Answer to Exercise 39 (on page 102)

```

1 const progress = document.querySelector("#progress");
2 const steps = document.querySelectorAll(".steps-section li");
3
4 for (const step of steps) {
5   step.addEventListener("click", () => {
6     step.classList.toggle("done");
7     const done = document.querySelectorAll(".steps-section li.done").length;

```

```
8     progress.textContent = done + " of " + steps.length + " steps done";
9   });
10 }
```

Counting the elements with the class `done` each time is simpler and safer than keeping a running total, which could get out of step with the page.

Answer to Exercise 40 (on page 102)

The count goes back to 0. The variable `likes` lives in the server's memory, which is wiped when the program stops. Save it in a file instead:

```
1  const fs = require("fs");
2
3  let likes = 0;
4  if (fs.existsSync("likes.txt")) {
5    likes = Number(fs.readFileSync("likes.txt", "utf8"));
6  }
```

Then, in the POST route, save the new count right after adding 1:

```
1  likes += 1;
2  fs.writeFileSync("likes.txt", String(likes));
```

A real site would store data like this in a database, like the ones in the Data Tables in SQL chapter.

Answer to Exercise 41 (on page 103)

The count goes up by 1,000. To the server, a request from `curl` looks exactly like a request from the button: both are just HTTP requests, and anything in the request can be faked. The server has to decide for itself whom to trust. Real sites make you log in and allow one like per account, or limit how many requests each address can send per minute.

Answer to Exercise 42 (on page 113)

A	B	out
0	0	1
0	1	0
1	0	0
1	1	0

It is the NOR gate: the output is 1 only when both inputs are 0.

Answer to Exercise 43 (on page 113)

NOT: $\text{NOR}(A, A) = \neg(A \text{ or } A) = \neg A$.

OR: a NOR followed by a NOR used as a NOT: $\text{NOR}(\text{NOR}(A, B), \text{NOR}(A, B)) = A \text{ or } B$. That is 2 NOR gates. (AND then follows from De Morgan's law: $A \text{ and } B = \text{NOR}(\neg A, \neg B)$.)

Answer to Exercise 44 (on page 113)

$1011_2 + 0110_2 = 10001_2$. In decimal, $11 + 6 = 17 = 16 + 1$. The largest 4-bit number is 15, so 17 does not fit; the final carry out becomes the fifth bit.

Answer to Exercise 45 (on page 114)

$A \oplus B = 0$, so $\text{sum} = 0 \oplus 1 = 1$. $C_{\text{out}} = (1 \text{ and } 1) \text{ or } (0 \text{ and } 1) = 1$. The carry and sum are 1 and 1: in binary, $1 + 1 + 1 = 11_2 = 3$.

Answer to Exercise 46 (on page 114)

$64 \times 9 = 576$ NAND gates, and $576 \times 4 = 2,304$ transistors. That is a tiny fraction of a modern chip.

Answer to Exercise 47 (on page 114)

```
1 def full_adder(a, b, carry_in):
```

```

2     n1 = nand(a, b)
3     s = nand(nand(a, n1), nand(b, n1))
4     m1 = nand(s, carry_in)
5     total = nand(nand(s, m1), nand(carry_in, m1))
6     carry_out = nand(n1, m1)
7     return total, carry_out

```

The test still reports 0 errors, and a 4-bit addition now uses $4 \times 9 = 36$ NAND gates. The trick works because, by De Morgan's law, $\text{NAND}(n_1, m_1)$ is $(A \text{ and } B)$ or $((A \oplus B) \text{ and } C_{in})$.

Answer to Exercise 48 (on page 115)

The leftmost full adder cannot finish until its carry in is known, which waits on the adder to its right, and so on down the line. In the worst case (for example, adding 1 to a number that is all 1s), the carry must pass through all 64 full adders, one after another. Real chips use *carry-lookahead* adders, which spend extra gates to figure out the carries in parallel.

Answer to Exercise 49 (on page 115)

```

1  def subtract(a_bits, b_bits):
2      carry = 1
3      result = []
4      for a, b in zip(a_bits, b_bits):
5          total, carry = full_adder(a, not_(b), carry)
6          result.append(total)
7      return result
8
9  print(to_int(subtract(to_bits(13, 4), to_bits(6, 4)))) # 7

```

It prints 7. The final carry is dropped, because a 4-bit two's complement answer has only 4 bits.



INDEX

- A* Algorithm, 8
- A* search, 8
- A-star Algorithm, 8
- acknowledgments, 22
- adjacency list, 5
- adjacency matrix, 6
- admissible heuristic, 10
- algorithmic reduction, 14
- API
 - calling from Python, 66
- API endpoint, 64
- API key, 80
- API keys, 80
- arithmetic logic unit, 112
- arrow function, 94
- authentication, 81
- authorization, 81
- Authorization header, 80

- bijection, 11
- binary
 - addition, 109
- Bipartite Graph, 17
- bipartite graph, 17
- bit
 - as voltage, 105
- border, 90

- cache, 28
- Chebyshev distance, 10
- class attribute, 34

- CMOS, 106
- collection, 83
- CRUD operations, 72
- CSS, 36, 87
 - box model, 90
 - cascade, 89
 - flexbox, 91
 - rule, 88
 - selector, 88
- cURL, 74
- cycle, 16
- Cycle Error, 16

- data attribute, 95
- De Morgan's laws, 108
- DELETE request, 79
- digital logic, 105
- Directed Acyclic Graph, 16
- directed adjacency matrix, 6
- distance-vector routing, 27
- DNS, 28
- DOCTYPE, 34
- Document Object Model, 93
- DOM, 93
- Domain Name System, 28

- endpoint, 72
- Euclidean distance, 9
- event listener, 93

- flexbox, 91

forwarding table, 25

full adder, 109

gate, 105

graph isomorphism, 12

half adder, 109

header, 21

heuristic, 9

HTML, 33

 attributes, 34

 div, 38

 elements of, 33

 forms, 54

 headings, 41

 images, 56

 links, 45

 lists, 45

 metadata, 37

 navigation, 49

 semantic elements, 39

 structure of, 34

 tables, 50

 text formatting, 43

HTTP, 63

HTTP request, 63

HTTP response, 65

id attribute, 34

injection, 11

internet, 21

inverter, 106

IP address, 23

IPv4, 23

IPv6, 24

Isomorphism, 12

JavaScript, 37, 87

JSON, 66

latency, 27

layer, 29

link-state routing, 27

logic gate, 107

margin, 90

matrix, 6

minimum spanning tree, 8

MOSFET, 105

NAND gate, 106

 universality, 108

NAT, 23

nearest neighbour, 14

network address translation, 23

network layers, 29

next hop, 25

packet, 21

packets, 21

padding, 90

PATCH request, 78

payload, 21

port, 24

POST request, 75

Postman, 82

prefix, 24

promise, 99

properties, 88

PUT request, 77

query string, 67

resolver, 28

responsive, 92

responsive design, 92

RESTful API, 71

ripple-carry adder, 110

root server, 28

router, 21

routers, 21

routing, 25

rules, 88

selector, 88

self-closing element, 33

spanning tree, 8

status code, 65

strongly connected, 20

subtraction

 in hardware, 112

surjection, 11

taxicab distance, 10

TCP, 22

time to live, 28

- topological order, [15](#)
- topological sort, [15](#)
- Topological Sorting, [15](#)
- transistor, [105](#)
- Transmission Control Protocol, [22](#)
- truth table, [105](#)
- truth tables, [105](#)

- UDP, [23](#)
- universal, [109](#)
- User Datagram Protocol, [23](#)

- Web APIs, [71](#)
- Web Development
 - HTML, [36](#)

- XOR, [107](#)