



CONTENTS

1	Hash Tables	3
1.1	Structure of a Hash Table	3
1.2	Inserting and Retrieving Data	3
1.3	Handling Collisions	3
1.4	Hashing Strings	4
1.5	Time Complexity	5
2	Recursion	7
2.1	What Recursion Means	7
2.2	Factorial	7
2.3	The Call Stack	8
2.4	Binary Search, Recursively	8
2.5	Recursive Traversal of a Tree	8
2.6	The Cost of Repeated Work	9
3	Trees	13
3.1	Binary Trees and Binary Search Trees	14
3.2	File Structures and File System Hierarchies	16
3.3	Balanced Trees: AVL and Red-Black	18
3.4	B-Trees, Tries, and Suffix Trees	19
4	Tree Traversal	21
4.1	BFS	21
4.1.1	Pseudocode	23

4.2	DFS	24
4.2.1	Pre-order	24
4.2.2	Post-order	25
4.2.3	In-order	25
4.2.4	Pseudocode	26
4.3	Recursive vs Iterative Traversal	26
4.3.1	Pre-order and Post-order Numbering	27
5	Finding Shortest Paths	29
5.1	Introduction — BFS revisited	29
5.2	Dijkstra's Algorithm	30
5.2.1	Algorithm Description	30
5.2.2	Implementation	33
5.3	Bellman-Ford Algorithm	36
6	Priority Queues	39
6.1	A Naive Implementation of the Priority Queue	39
6.2	Using the Priority Queue	40
6.3	Binary Search	42
6.4	Algorithm	43
7	Dynamic Programming	45
7.1	Remember Fibonacci?	45
7.2	Longest Common Subsequence	46
7.2.1	Thinking Recursively	46
7.2.2	Filling the Table	47
7.2.3	The Code	48
7.3	Building a Diff	49
7.4	The Dynamic Programming Recipe	50
A	Answers to Exercises	55
	Index	63

Hash Tables

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

Think about a school locker room. Each student has a locker number, and that number comes from a rule, such as the last two digits of the student ID. To find your locker, you do not search every row. You apply the rule and go straight to the right locker. A hash table works the same way.

1.1 Structure of a Hash Table

A hash table is composed of an array (the “table”) and a hash function. The array has a predetermined size, and each location (or “bucket”) in the array can hold an item (or several items if collisions occur, as will be discussed later). The hash function is a function that takes a key as input and returns an integer, which is then used as an index into the array. In this chapter, we will use the simple hash function $h(k) = k \bmod 5$ for a table with 5 slots.

1.2 Inserting and Retrieving Data

When inserting a key-value pair into the hash table, the hash function is applied to the key to compute the index for the array. The corresponding value is then stored at that index.

When retrieving the value associated with a key, the hash function is applied to the key to compute the array index, and the value is retrieved from that index.

Python’s dict is a hash table, so the same idea is behind the dictionaries you already use.

1.3 Handling Collisions

A collision occurs when two different keys hash to the same index. There are several methods for handling collisions:

- **Chaining (or Separate Chaining):** In this method, each array element contains a linked list of all the key-value pairs that hash to the same index. When a collision occurs, a new key-value pair is added to the end of the list.
- **Open Addressing:** In this method, if a collision occurs, we look for another open slot in the array and store the key-value pair there. One simple version is **linear probing**, which checks the next slot, then the one after that, and so on, wrapping around to the start of the array. When looking up a key, we keep checking slots until we find the key or reach an empty slot.

Exercise 1 Chaining with a Hash Function

Use $h(k) = k \bmod 5$ and chaining. Insert the keys 12, 7, 17, and 3, in that order. Show the contents of each bucket, 0 through 4.

Working Space

Answer on Page 55

Exercise 2 Linear Probing

Use $h(k) = k \bmod 5$ and linear probing in an array of 5 slots. Insert the keys 12, 7, 17, and 3, in that order. Show the contents of slots 0 through 4.

Working Space

Answer on Page 55

1.4 Hashing Strings

Keys are often strings, not numbers. A hash function for strings turns the characters into one number. A simple (but weak) one adds the character codes: "cat" becomes $99 + 97 + 116 = 312$. Its flaw is that "act" and "tac" get the same number. Real hash functions mix the characters so that small changes scatter results all over the table. Python's built-in hash does this.

Here is a complete hash table with chaining, in Python:

```

1 class HashTable:
2     def __init__(self, size=8):
3         self.buckets = [[] for _ in range(size)]
4
5     def _index(self, key):
6         return hash(key) % len(self.buckets)
7
8     def put(self, key, value):
9         bucket = self.buckets[self._index(key)]
10        for i, (k, v) in enumerate(bucket):
11            if k == key:          # key already here: replace value
12                bucket[i] = (key, value)
13            return
14        bucket.append((key, value))  # new key: add to the chain
15
16    def get(self, key):
17        for k, v in self.buckets[self._index(key)]:
18            if k == key:
19                return v
20        return None
21
22 ages = HashTable()
23 ages.put("Maya", 16)
24 ages.put("Jonah", 15)
25 ages.put("Maya", 17)          # Maya had a birthday
26 print(ages.get("Maya"))      # 17
27 print(ages.get("Jonah"))     # 15
28 print(ages.get("Priya"))     # None

```

As a table fills up, its chains get longer and lookups slow down. Real hash tables (including Python's dict) watch the *load factor*, the number of keys divided by the number of buckets. When it gets too high, they allocate a bigger array and reinsert every key.

1.5 Time Complexity

In an ideal scenario where hash collisions do not occur, hash tables achieve constant time complexity $O(1)$ for search, insert, and delete operations. However, due to hash collisions, the worst-case time complexity can become linear $O(n)$, where n is the number of keys inserted into the table.

Using good hash functions and collision resolution strategies can minimize this issue and allow us to take advantage of the hash table's efficient average-case performance.

Exercise 3 Worst Case

Suppose a hash function sends every key to bucket 0. Using chaining, how long does it take to look up a key that was inserted last, if there are n keys in the table? Why does this show that the worst case is $O(n)$?

Working Space

Answer on Page 55

Recursion

2.1 What Recursion Means

A *recursive* function is a function that calls itself. Think of Russian nesting dolls. To find the smallest doll, you open the big doll, then open the next one, and so on, until you reach a doll that does not open. Many problems can be split into a smaller copy of the same problem. Recursion solves the big problem by solving the smaller one and then combining the answers.

Every recursive function needs two parts:

- A *base case*, which returns an answer directly without calling the function again.
- A *recursive case*, which calls the function on a smaller input.

If the base case is missing, or if the recursive case never gets closer to it, the function calls itself forever.

2.2 Factorial

The factorial $n!$ is $n \cdot (n - 1)!$, with $0! = 1$. That definition is recursive, and the code is almost the same:

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 print(factorial(5)) # 120
```

The base case is $n == 0$. Each call makes the input one smaller, so it reaches the base case after n calls.

2.3 The Call Stack

Each time a function is called, Python stores a *frame* on the call stack. The frame holds that call's variables. When the call returns, its frame is removed. For `factorial(3)`, the stack grows to four frames: `factorial(3)`, `factorial(2)`, `factorial(1)`, and `factorial(0)`. Then the answers return back up the stack: 1, 1, 2, and 6.

2.4 Binary Search, Recursively

Binary search finds an item in a *sorted* list the way you look up a word in a dictionary: open to the middle, decide which half the word must be in, and repeat on that half. Each step throws away half of what is left, so even a million items take only about 20 steps. It is naturally recursive. Look at the middle item. If it matches, return its index. Otherwise, search only the half that could contain the target.

```
1 def binary_search(items, target, low, high):
2     if low > high:
3         return -1
4     mid = (low + high) // 2
5     if items[mid] == target:
6         return mid
7     elif items[mid] < target:
8         return binary_search(items, target, mid + 1, high)
9     else:
10        return binary_search(items, target, low, mid - 1)
11
12 data = [1, 3, 5, 7, 9, 11]
13 print(binary_search(data, 7, 0, len(data) - 1)) # 3
14 print(binary_search(data, 4, 0, len(data) - 1)) # -1
```

The first base case, `low > high`, handles a target that is not in the list. The second, `items[mid] == target`, handles a match.

2.5 Recursive Traversal of a Tree

A *tree* is a structure like a family tree or the folders on your computer: each node can have children, which can have children of their own. (You will study trees in detail in the next chapter.) A tree is naturally recursive: each child is the root of a smaller tree. To visit every node, visit the node, then visit each of its subtrees.

```
1 class Node:
2     def __init__(self, name, children=None):
3         self.name = name
```

```

4         self.children = children or []
5
6     def visit(node):
7         names = [node.name]
8         for child in node.children:
9             names.extend(visit(child))
10        return names
11
12    tree = Node('a', [Node('b', [Node('d')]), Node('c')])
13    print(visit(tree)) # ['a', 'b', 'd', 'c']

```

A node with no children is the base case. Its loop does nothing, so it returns just its own name.

2.6 The Cost of Repeated Work

The Fibonacci numbers are defined by $F(n) = F(n - 1) + F(n - 2)$, with $F(0) = 0$ and $F(1) = 1$. A direct recursive version is short:

```

1     def fib(n):
2         if n < 2:
3             return n
4         return fib(n - 1) + fib(n - 2)

```

It is correct, but it is slow. The call for `fib(10)` makes 177 calls in total, and `fib(15)` makes 1,973. The same smaller values are computed again and again. The number of calls grows exponentially with n .

Memoization fixes this by saving each answer the first time it is computed:

```

1     def fib_memo(n, memo=None):
2         if memo is None:
3             memo = {}
4         if n < 2:
5             return n
6         if n not in memo:
7             memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)
8         return memo[n]
9
10    print(fib_memo(80)) # 23416728348467685

```

Saving answers to smaller problems so they are never computed twice is the central idea of *dynamic programming*, the subject of its own chapter later in this volume.

Exercise 4 **Trace the calls**

Write out every call made by `factorial(3)`, in order, and then the value returned by each call.

Working Space

Answer on Page 56

Exercise 5 **Sum a list**

Write a recursive function `total(items)` that returns the sum of a list. What is its base case?

Working Space

Answer on Page 56

Exercise 6 **Counting calls**

How many calls does `fib(5)` make in total? Use the formula that the number of calls for `fib(n)` is $2F(n+1) - 1$, where F is the Fibonacci sequence, and check the answer by counting the calls.

Working Space

Answer on Page 56

Exercise 7 **Missing base case**

What happens when `factorial(-1)` is

Working Space

called, using the code above?

Answer on Page 56

Trees

Trees are one of the most versatile and widely used data structures in computer science. A *tree* is a hierarchical data structure consisting of nodes, where each node has a value and a set of references (*edges*) to its child nodes. The node at the top of the hierarchy is called the *root*, and nodes with the same parent are called *siblings*, as seen in Figure 3.1.

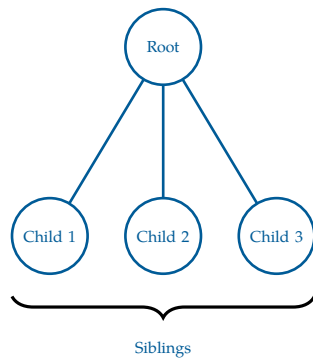


Figure 3.1: A tree with a root node and 3 children.

Trees are a special kind of graph, a data structure comprised of nodes and edges. A tree is a graph with two properties:

- The graph is connected, such that there exists a path between every pair of nodes
- The graph is acyclic; no cycles exist in the graph

With this comes the caveat that the number of edges in a tree is $\text{edges} = n - 1$, where n is the number of nodes in the tree. This is because a tree with n nodes must have exactly $n - 1$ edges to maintain its connected and acyclic properties.

If there were more than $n - 1$ edges, it would create a cycle, violating the acyclic property. If there were fewer than $n - 1$ edges, it would not be fully connected, violating the connected property.

Less formally, a *forest* is a collection of multiple trees (may be disconnected), and a *tree* is a forest with only one tree. Lots of agricultural terms in this chapter!

Trees have a unique structural property: between any two nodes, there exists exactly one simple path. This follows directly from the fact that trees are both connected and acyclic. As a result, there is only one way to move from a start node to an end node.

Although trees are often drawn with a root at the top, this is a matter of convention. Any node in a tree can be chosen as the root, and doing so induces a hierarchical structure in which edges are interpreted as parent-child relationships.

Once a root is selected, every node (except the root) has exactly one parent, and zero or more children. Because of this relationship, any randomly selected node can form a tree (we call this a *subtree*, consisting of the node and all its descendants), making the tree structure *recursive*. A function that works on a tree can call itself on each subtree, and

each call handles a smaller tree.

Nodes with no children are called, as you would expect, *leaves*, as they are the outermost node of a tree. The *depth* of a chosen node is the number of edges from the *root* to that node, and the *height* of a tree is the maximum depth among all nodes. Equivalent to the height of a tree is the maximum number of edges from the *root* to any leaf. You may see the *level* of a node referred to as the number of edges from the *root* to that node, similar to depth. By convention, the level and depth of the root is 0. See Figure 3.2.

Some more relevant terminology you may encounter in the context of trees:

- *Path*: A sequence of nodes where each adjacent pair is connected by an edge. This can be used to show the idea of how to get between two nodes in a tree, for example, between two cities in a flight path network.
- *Ancestor*: A node that lies on the path from the root to a given node. This is done by repeatedly traversing from a child node to a parent node until the selected node is reached.
- *Descendant*: Any node you can reach from a given node by repeatedly moving from a parent to one of its children. A node's descendants are everything in its subtree except itself.

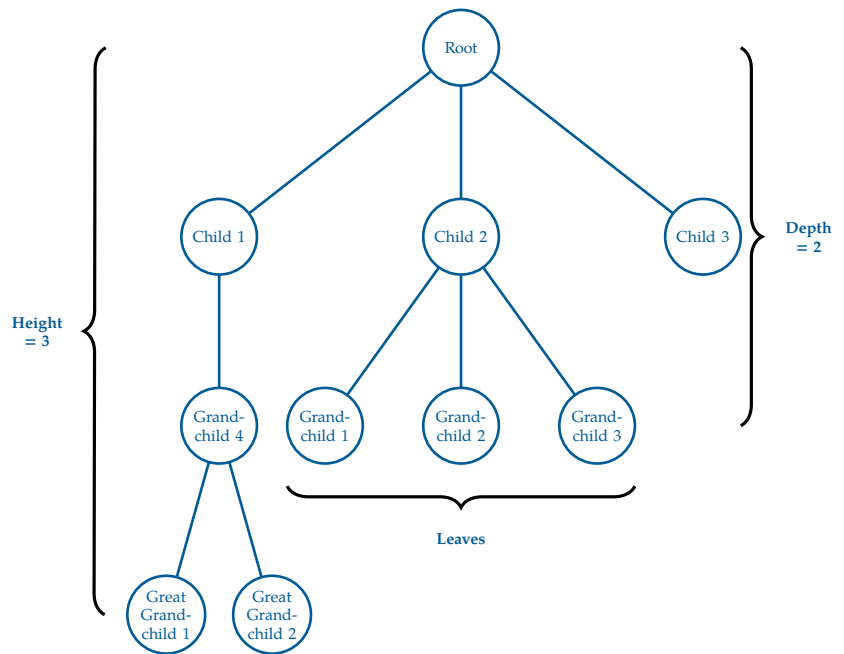


Figure 3.2: A tree with a root node and 3 children, a few grandchildren, and 2 great grandchildren.

3.1 Binary Trees and Binary Search Trees

The most common type of tree is the *binary tree*, where each node has *at most* two children, referred to as the *left child* and the *right child*. Binary trees are widely used in various applications, including expression parsing, binary search trees, and heaps.

Let us put this idea into practice. Recall our binary search implementation from the recursion chapter. Given a chosen word and a sorted list of words, such as a dictionary, we can find the page of the chosen word by repeatedly dividing the available pages in half and comparing the elements on the page to the chosen word. This is much easier than, for example, going through every single page and checking if the word is on that

page.

Say we wanted to represent our dictionary as a tree that we could recursively binary search through. For simplicity, we will limit our words to mango, apple, pear, banana, anagram, peony and peach. These 7 words form a binary tree of height 2, as seen in Figure 3.3. We select the word mango as the root as it is the middle word after sorting alphabetically.

Similar to binary search, we can find any word in the tree by starting at the root and comparing the target word to the current node's word. If the target word is less than the current node's word, we move to the left child; if it is greater, we move to the right child. We repeat this process until we find the target word or reach a leaf node, which tells us that the target word is not in the tree. This structure allows us to cut the available words in half with each step in the tree we go.

What if we want to insert into a binary search tree? We must complete four actions:

- Find the correct location for the new node.
- Create and insert the node at that location,
- Connect it to its respective parent(s).
- Ensure the BST properties are maintained.

The new node is always a leaf. We find its location the same way we search: at each node, go left if the new word comes first alphabetically, and go right if it comes later.

Let us say, on our tree above, we want to add orange to the tree. orange comes after mango, so we go right to pear. orange comes before pear, so we go left to peach. orange comes before peach, and peach has no left child. So we make orange the left child of peach.

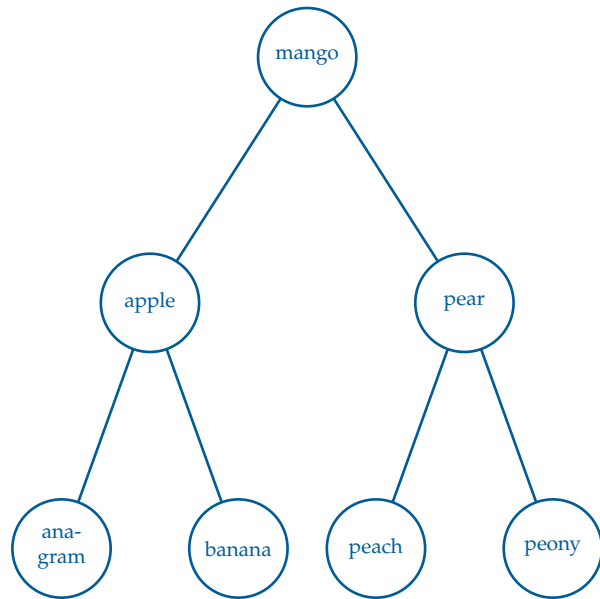


Figure 3.3: A binary search tree with 7 words.

Exercise 8 Insert into a Search Tree

Using the tree in Figure 3.3, insert kiwi. Write the sequence of words you compare kiwi with, and say where the new node goes.

Working Space

Answer on Page 56

3.2 File Structures and File System Hierarchies

Computers have always needed a way to store and organize data. Prior to computers, many systems of organization were used to store physical data, such as filing cabinets, libraries, and warehouses.

These systems often had a hierarchical structure, with a top-level category that branched out into subcategories, with these subcategories further branching out into more specific categories, and so on.

For example, a library might have two top-level categories: fiction and non-fiction. Within the category non-fiction, you may find psychology, biographies, autobiographies, self-help, journalism books, or even nature guides. Under the category fiction, you may find romance, classics, crime, science fiction, thriller, or fantasy novels. Of course, these sub-genres can have even further categorizations too.

A hospital may have patient records categorized by first letter of last name, and then sorted alphabetically from there.

With the development of the computer, sorting these types of hierarchical data became easy with the concept of directories. A *directory*, also called a *folder*, is a categorical way of organizing data (for example, by storing files or other subdirectories). We will stick to the more technical term of directories, however, any time you create a folder on your computer, you are making a *directory*. The name comes from how, in the twentieth-century, many phone books were referred to as telephone directories.

This structure naturally fits into the tree data structure:

- The **root** of the tree is the top-level directory (/ on Linux/Unix¹ systems or C:\ on

¹There is a very helpful video in Linux File Systems in your digital resources. Check it out!

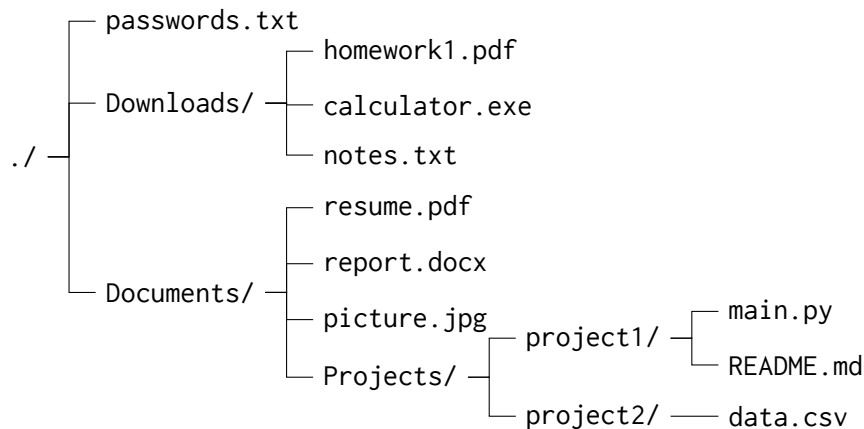
Windows).

- A **node** naturally fits the idea of either *directories nodes* or *files nodes*.
- A **directory node** may have children, where its children are subdirectories or files.
- A **file node** is a **leaf**, meaning it has no children.

The **path** to a file describes the unique sequence of directories from the root to that file.

Example

A small file system might look like:



In tree terminology:

- `./` is the root node.
- Downloads, Documents, and its subdirectories are internal nodes (directories).
- All files, such as those ending in `.txt`, `.png`, `.exe`, `.csv`, `.docx`, `.py` files are leaf nodes.

Side note: please do not store your passwords in a plain text file!

Exercise 9 File Paths

In the file system tree above, what is the path to `data.csv`? What is its depth?

Working Space

Answer on Page 56

3.3 Balanced Trees: AVL and Red-Black

A binary search tree is fast only if it is bushy. Insert the words anagram, apple, banana, mango, peach, pear, peony *in alphabetical order*, and each new word goes to the right of the last one. The “tree” is just a chain of 7 nodes with height 6, and searching it is no faster than reading a list from the top. The same seven words in Figure 3.3 have height 2.

A *balanced* tree keeps its height close to $\log_2 n$ no matter what order the data arrives in. Two famous designs:

- An *AVL tree* (named for its inventors, Adelson-Velsky and Landis, 1962) requires that at every node, the heights of the left and right subtrees differ by at most 1. After an insertion breaks that rule, the tree performs a *rotation*: it lifts a child up into its parent’s place, shifting a few pointers. A rotation keeps the left-to-right (alphabetical) order intact while making the tree shorter.
- A *red-black tree* colors every node red or black and enforces a few rules: the root is black, a red node never has a red child, and every path from a node down to an empty spot passes through the same number of black nodes. Together, these rules guarantee that no path is more than twice as long as any other. Red-black trees are slightly less strictly balanced than AVL trees but need fewer rotations, which is why C++’s `std::map` and Java’s `TreeMap` use them.

Either way, searching, inserting, and deleting take $O(\log n)$ time: a million items need only about 20 levels.

Exercise 10 How Tall?

Working Space

1. A binary tree with every level completely full has height h . How many nodes does it have?
2. About how tall is a balanced binary search tree holding 1,000,000 words? How tall could an unbalanced one be?

Answer on Page 57

3.4 B-Trees, Tries, and Suffix Trees

Not every tree is binary.

A *B-tree* lets each node hold many keys (often hundreds) and have many children. Reading from a disk is slow, but reading a big chunk is not much slower than reading a small one. So databases and file systems store each node in one disk block. A B-tree holding a billion records may be only 4 or 5 levels deep, which means only 4 or 5 slow disk reads per lookup.

A *trie* (from *retrieval*, usually pronounced “try”) stores words letter by letter. Each edge is a letter, and each path from the root spells the start of a word. Words that share a prefix share a path: car, cart, and cat all start down the c–a branch. To autocomplete ca, your phone walks two edges and lists everything below that node.

A *suffix tree* is a compressed trie of all the suffixes of one long string. For the string banana, those are banana, anana, nana, ana, na, and a. Every substring is the beginning of some suffix, so you can check whether any pattern appears in the text by walking down from the root, in time proportional to the length of the pattern, not the text. Biologists use suffix trees to search DNA, where the “text” can be billions of letters long.

Exercise 11 Build a Trie

Draw a trie for the words to, tea, ten, and in. How many nodes does it have, counting the root?

Working Space

Answer on Page 57

Tree Traversal

Trees can be “traversed” in a few different ways. When we say traversing a tree, we mean that we are visiting and extracting data from each node in a tree. We classify these *traversals* based on the order in which the nodes are visited.

This chapter will discuss *breadth-first search*, also known as *BFS*, which traverses in a *level-order*. Each level is traversed in sequential order.

Alternatively, there is *depth-first search* (*DFS*), which travels as far down a branch as possible before backtracking and checking other branches. Within DFS there are 3 common traversal methodologies: *in-order*, *pre-order* and *post-order*.

While the demonstrations for these traversals will be for trees, they work the same for any given graph, under one condition: using DFS or BFS on a *graph* requires that you have a preselected node as your *start node*.

4.1 BFS

Let us look at running BFS on a tree. Starting out with this tree:

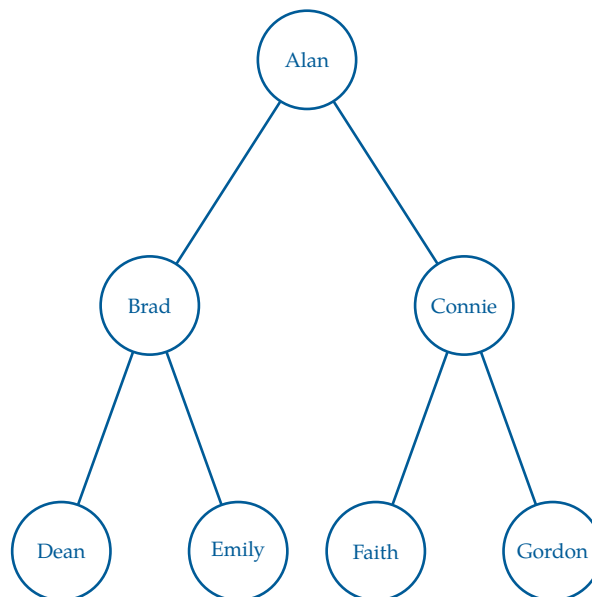


Figure 4.1: Our original tree containing a list of names to be traversed.

To perform BFS, we use a *queue*, which follows a first-in, first-out (*FIFO*) order. We begin by placing the root node into the queue. At each step, we remove the front node from the queue, visit it, and then add all of its children to the *back* of the queue. (Queues are from the Stacks and Queues chapter.)

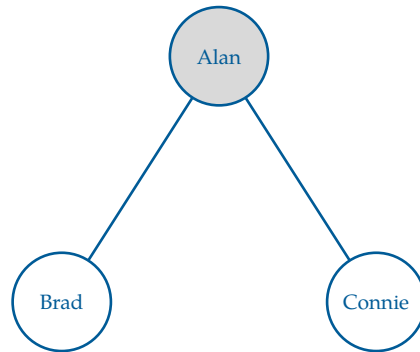


Figure 4.2: Step 1: Visit Alan, then add its children (Brad, Connie) to the queue.

Here, we visit the root node first, which is Alan. After visiting Alan, we add its children, Brad and Connie, to the queue. Since there are no other nodes to visit at this level, we move on to the next level of the tree.

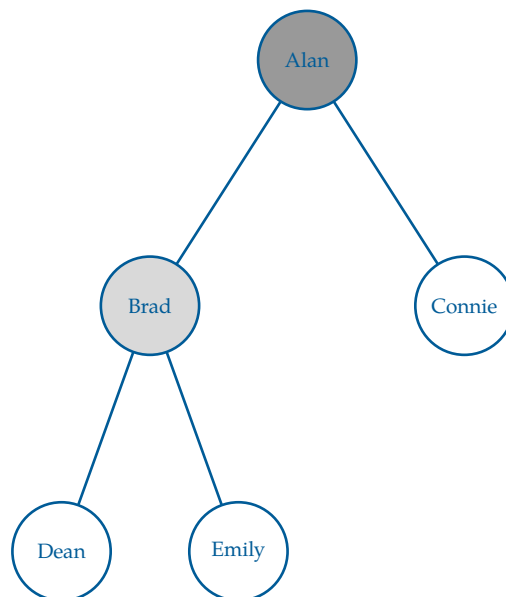


Figure 4.3: Step 2: Visit Brad, then add its children (Dean, Emily) to the queue.

After visiting Brad, the queue holds Connie, Dean, and Emily. Next we visit Connie and add her children, Faith and Gordon, to the back. Here is a step trace of the BFS traversal:

Step	Queue	Output (visited)
Start	[Alan]	[]
1	[Brad, Connie]	[Alan]
2	[Connie, Dean, Emily]	[Alan, Brad]
3	[Dean, Emily, Faith, Gordon]	[Alan, Brad, Connie]
4	[Emily, Faith, Gordon]	[Alan, Brad, Connie, Dean]

Continuing this process until the queue is empty gives the final traversal:

Alan, Brad, Connie, Dean, Emily, Faith, Gordon

This ordering reflects a level-by-level traversal of the tree, which is why BFS on a tree is also called level-order traversal.

To use BFS for *searching* rather than full traversal, we simply introduce a target value and stop the algorithm as soon as the target is found. Instead of visiting every node, we check each node when it is dequeued and terminate early if it matches the desired value.

For example, suppose we want to find “Faith” in the tree. BFS will examine nodes level by level: Alan, then Brad and Connie, then Dean, Emily, and Faith. As soon as Faith is encountered (we *dequeue* it from the queue), the search stops, without visiting any remaining nodes.

4.1.1 Pseudocode

Our steps can be reflected as pseudocode as follows:

Algorithm 1 Breadth-First Search (BFS) - Iterative

```

procedure BFS(root, target)
  Q = empty queue
  ENQUEUE(Q, root)
  while Q is not empty do
    node = DEQUEUE(Q)
    if node = target then
      return node
    end if
    for each child of node do
      ENQUEUE(Q, child)
    end for
  end while
  return null
end procedure

```

This procedure is often used in game applications to find paths, such as maze solvers, chess algorithms, and tic-tac-toe solutions. It is also used in social network analysis to find the shortest path between two users, and in web crawling to explore the structure of websites.

4.2 DFS

In contrast, we can run a *depth-first search* on the original tree (see Figure 4.1). Depth-first search, as the name implies, goes as deep down the tree until reaching a leaf, and then backtracks to the next possible, unsearched path. DFS (implicitly) uses a stack to explore the graph or tree.

The steps are fairly simple:

1. Push the root node into the stack.
2. Pop a node from the stack, and mark it as visited.
3. Push the node's children onto the stack.
4. Repeat steps 2 and 3 until the stack is empty.

There are 3 orders in which we can traverse the whole tree. We will look at pre-order, post-order, and in-order.

However, these orders essentially perform the same basic three tasks,

V Visit the selected node.

L Run DFS on the selected node's left subtree.

R Run DFS on the selected node's right subtree.

The recursion terminates when we reach a leaf. At this point, the algorithm returns to the previous call, which is how *backtracking* occurs. This backtracking is done via the stack we had mentioned previously.

4.2.1 Pre-order

In pre-order traversal, we visit the node *before* exploring its subtrees. The order is:

(V, L, R)

We visit the node *before* its children because the node's value is needed *prior* to exploring its subtrees. This is useful when the node represents a decision that affects which branch of the tree you go down, such as in tree copying or directory traversal like we talked about in the previous chapter.

Using our original tree, the pre-order results are:

Alan, Brad, Dean, Emily, Connie, Faith, Gordon

4.2.2 Post-order

In post-order traversal, we visit the node *after* exploring both subtrees. The order is:

(L, R, V)

We visit the node *after* its children because the node's computation depends on results from both subtrees. This is useful when combining subtree results or deleting trees, where children must be handled before the parent.

The post-order traversal of our original tree is

Dean, Emily, Brad, Faith, Gordon, Connie, Alan

4.2.3 In-order

In in-order traversal, we visit the node *between* exploring its subtrees. The order is:

(L, V, R)

We visit the node *between* its subtrees because the correct interpretation of the node depends on partially processed children. In-order makes sense only for binary trees, since it needs a left and a right subtree to put the node between. On a binary search tree, in-order visits the keys in *sorted order*: run it on the word tree from the previous chapter and you get the words alphabetically.

The in-order traversal of our original tree becomes

Dean, Brad, Emily, Alan, Faith, Connie, Gordon

4.2.4 Pseudocode

The previous sections (pre-order, in-order, and post-order) describe *when* a node is visited during a depth-first traversal.

The following algorithm instead describes the general *mechanism* of depth-first search. This implementation corresponds to a pre-order traversal. The right child is pushed first, so the left child is popped first. Other traversal orders (in-order and post-order) require additional state and cannot be obtained by simple reordering alone.

Algorithm 2 Depth-First Search (DFS) - Iterative on a Tree

```
procedure DFS(root)
  S = empty stack
  PUSH(S, root)
  while S is not empty do
    node = POP(S)
    visit(node)                                ▷ V
    if node.right ≠ null then
      PUSH(S, node.right)                      ▷ R
    end if
    if node.left ≠ null then
      PUSH(S, node.left)                       ▷ L
    end if
  end while
end procedure
```

DFS is particularly useful for solving problems like connected-component detection in graphs and maze-solving.

4.3 Recursive vs Iterative Traversal

As we have established, trees are recursively defined structures, and as such, our algorithms for them can be defined as such too. This allows us to implement searches and traversals in two ways: *recursive* and *iterative*. Let us look at DFS recursively.

Importantly, they do not change the traversal order (pre-, in-, post-order); they only change how the traversal is implemented.

A *recursive traversal* relies on the program's *call stack* (recall it from the Introduction to C and Recursion chapters). Each time the algorithm explores a subtree, it makes a function call, and the system automatically remembers where to return when that call finishes. This matches the recursive definition of a tree, where each subtree is itself a tree. This allows us to get rid of the stack in DFS and solely rely on a computer's call stack.

Exercise 12 Recursive DFS

Take a look at the Iterative DFS code above.

How can we rewrite the pseudocode recursively, taking advantage of the tree's recursive structure? Try pre-order recursively first. Assume the tree is a binary tree.

Working Space

Answer on Page 57

Here is the advantage of the recursive version: to get in-order or post-order, just move the visit line (L, V, R or L, R, V). The iterative version cannot be changed that simply.

What about BFS? BFS is typically implemented iteratively because it relies on a queue. While recursive versions exist, they still depend on a queue and offer little conceptual advantage, so we omit them here.

4.3.1 Pre-order and Post-order Numbering

There also exists pre-order and post-order numbering, regardless of which version of DFS is run. Pre-order and post-order numbering are techniques used during a depth-first search (DFS) traversal to record when each node is encountered and when it is fully processed. A *pre-order number* is assigned when the traversal first visits a node, before exploring any of its children. A *post-order number* is assigned after all of the node's descendants have been explored and the traversal is about to return to the node's parent. These numbers can be viewed as *timestamps* that describe the progress of the DFS.

Pre-order numbers capture the order in which nodes are discovered, while post-order numbers capture the order in which nodes are completed. This information is useful in many graph and tree algorithms, including dependency analysis, topological sorting (which we will talk about in the future), and determining ancestor-descendant relationships.

Exercise 13 DFS Pseudocode

Working Space

Alter the *recursive* DFS pseudocode to include an assignment of pre-order and post-order numbers for every node. Use one shared counter that starts at 1.

Answer on Page 58

Exercise 14 **Pre- and Post-order numbers**

Working Space

Write the pre and post order numbers for the person graph we have been working with, Figure 4.1

Answer on Page 58

Finding Shortest Paths

5.1 Introduction — BFS revisited

We have seen BFS as a way to search graphs (or trees). It is a pretty simple process:

- select a start node
- move one distance from the root, and visit all nodes (v) on that level
- when visiting, queue all child nodes (v') of that node.
- keep going until either all nodes are marked *visited* or a provided *target node* is reached.

There is an issue here: what if the graph is weighted, and the paths are not equally weighted? For example, see this graph:

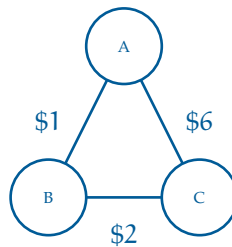


Figure 5.1: A graph of 3 nodes that would fail BFS.

BFS, here, would prefer $A \rightarrow C$, since it has an edge count of 1, whereas $A \rightarrow B \rightarrow C$ has an edge count of 2. However, the true shortest path, by weight, is $A \rightarrow B \rightarrow C$, with a weight of 3.

So BFS has a major limitation: it cannot handle weighted graphs. This is why we started it out on trees; typically trees do not have a set edge weight, so we were not limited by the order in which it expands.

To handle weighted graphs, we use a new algorithm that expands off of BFS, called *Dijkstra's Algorithm*. Dijkstra's Algorithm avoids the limitation of weights that BFS had by enqueueing the nodes in order of current shortest known distance.

5.2 Dijkstra's Algorithm

Edsger W. Dijkstra (pronounced *DYKE-strah*) was a great Dutch computer scientist. He came up with an algorithm for finding the cheapest path through a graph with weighted edges. Today it is known as *Dijkstra's Algorithm*. It is used in a wide variety of common problems, and is also both simple and elegant.

5.2.1 Algorithm Description

You are going to mark each node with how much it would cost to get there from some chosen origin node. For example, if you are shipping a container from Long Beach, you will mark each city with the cost of getting the container to that city.

You start by marking the price for Long Beach to zero (the container is already there). You then mark each adjacent city with the cost on the edge (figure 5.2a). Next, you declare Long Beach to be “visited” (figure 5.2b).

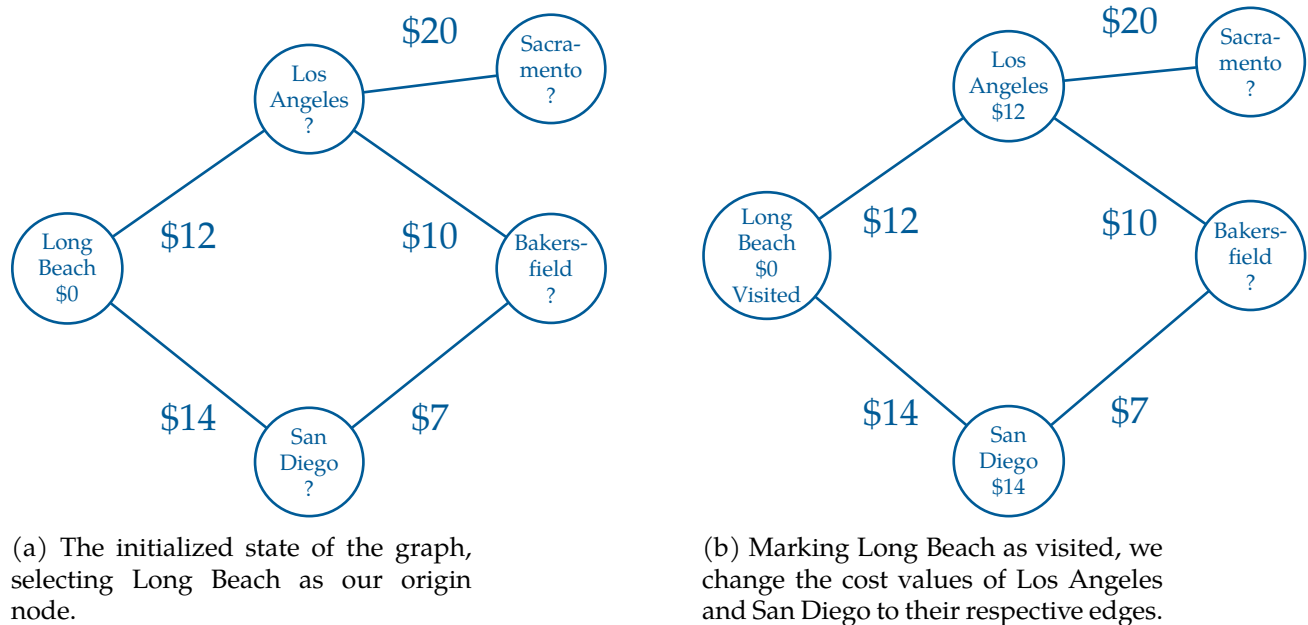


Figure 5.2: A map of weighted cities.

After that, you find the cheapest of the unvisited nodes. In this case, Los Angeles is cheaper than San Diego, so that is the node you will visit next.

You mark all of the unvisited nodes adjacent to Los Angeles, with the price to ship it to Los Angeles plus the cost of shipping the container from Los Angeles to that city (figure

5.3a). Note that Bakersfield is marked with \$22.

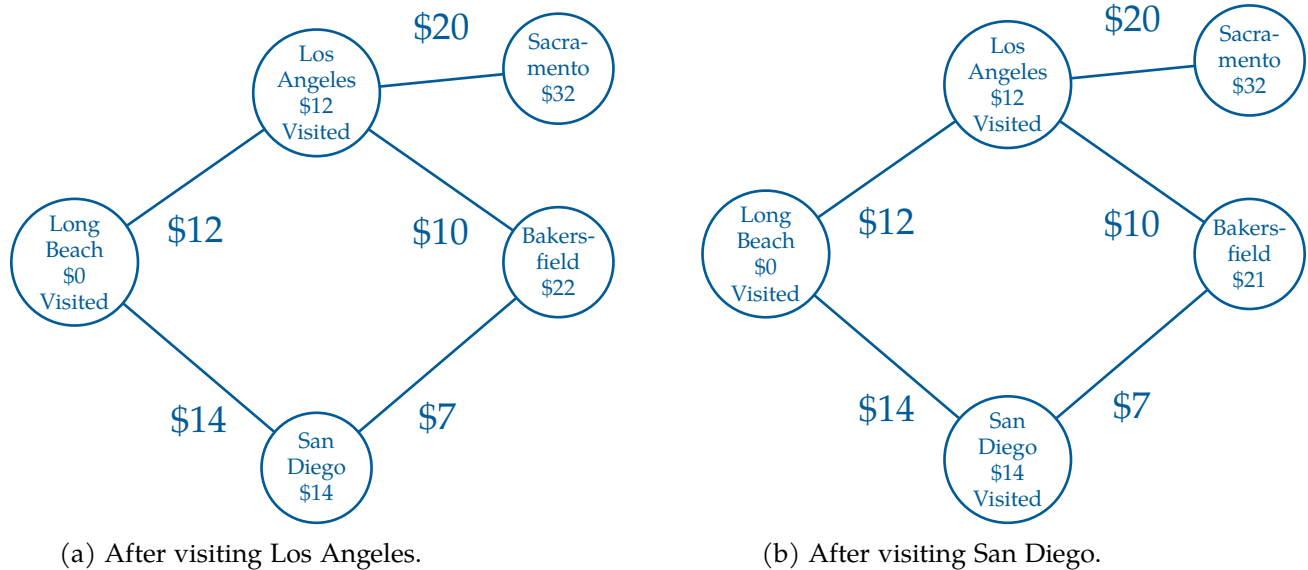


Figure 5.3: Visiting the next two minimum cost cities.

Now, the cheapest unvisited node is San Diego, so you mark its neighbors with the cost to ship to San Diego, plus the price to ship from San Diego to the neighbor (figure 5.3b).

Notice that Bakersfield is already labeled with \$22 from a route through Los Angeles. But the price would be \$21 if you shipped it to Bakersfield via San Diego. Because the new route is cheaper, you change the price to the lower value.

(What does it mean that a node is “visited”? If a node is marked visited, it is marked with a price that will not get any smaller.)

You continue visiting the cheapest unvisited node until all the nodes have been visited. Then you know every node has been marked with its lowest price.

In a big graph, each node may be marked several times in this process — each time with a lower price from a cheaper route.

Of course, once you have the price, you will ask, “What is the route that gets me that price?” So, we will also mark each node with the neighbor from which it would receive the shipment — the previous node. This is easy to do as we execute the algorithm.

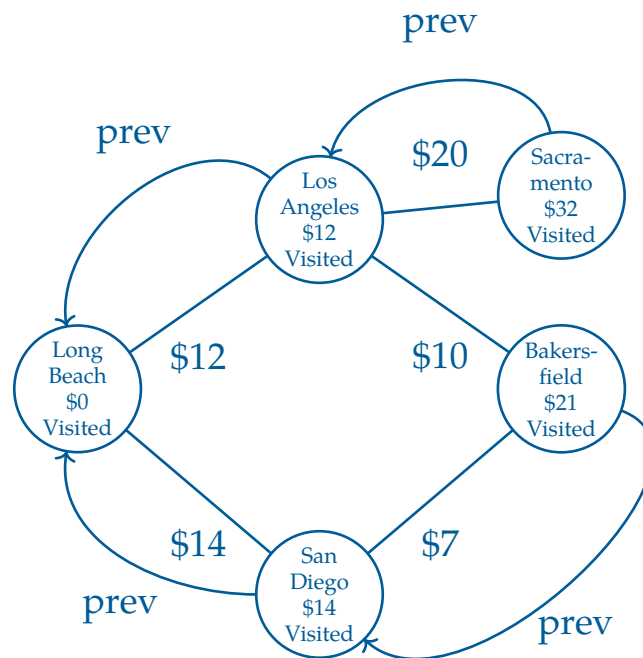


Figure 5.4: Marking each node with a previous pointer.

Now, to figure out the cheapest route from Long Beach to Bakersfield, we start at the destination and follow the prev pointer back through San Diego, then to Long Beach.

Exercise 15 Expanding on the Delivery Graph

Notice that the path between Los Angeles to Bakersfield is not used once we add in our previous labels. Why might this be?

Working Space

Answer on Page 58

Exercise 16 Dijkstra's Algorithm by Hand

Run Dijkstra's algorithm from A on this weighted undirected graph: A–B costs

Working Space

2, A–C costs 5, B–C costs 1, C–D costs 2, and B–D costs 6. Give the final cost to each node, and the cheapest path from A to D.

Answer on Page 59

5.2.2 Implementation

We do not actually want to sully our graph objects with the three additional pieces of information we need:

- The current minimal cost from the origin node. This is usually called the *dist*, from “distance”.
- The neighbor who gives us the current minimal cost. This is usually called *prev*, from “previous”.
- Whether the city is visited or not.

So, we will keep them in collections external to the graph.

For example, to keep track of the *dist*, we will have a *dist* dictionary. Each node will be a key, the current minimal cost will be the value. If the node has not received even a first cost, we will put in infinity as the cost.

(After the algorithm is run, if the cost of a node is still infinity, that means that it cannot be reached from the origin node.)

We will also have a *prev* dictionary. The final node will be the key, and its previous neighbor will be the value.

Finally, the graph has a list of all the nodes, so we can just keep a set of the unvisited nodes.

Add a method to the *Graph* class that implements Dijkstra’s algorithm:

```

1  def cost_from_node(self, origin_node):
2      # Cost of cheapest path from origin node discovered so far
3      # Initially the origin is zero and all the other are infinity
4      dist = {k: math.inf for k in self.nodes}
```

```

5     dist[origin_node] = 0.0
6
7     # The previous city on that cheapest path
8     prev = {}
9
10    # All the nodes start as unvisited
11    unvisited = set(self.nodes)
12
13    # While there are still unvisited nodes
14    while unvisited:
15
16        # Find unvisited node with lowest cost
17        min_cost = math.inf
18        for u in unvisited:
19            if dist[u] < min_cost:
20                current_node = u
21                min_cost = dist[u]
22
23        # If none are less than inf, we are done
24        # This happens in graphs that are not connected
25        if min_cost == math.inf:
26            return (dist, prev)
27
28        # Remove the lowest cost node from the unvisited list
29        unvisited.remove(current_node)
30
31        # Update all the unvisited neighbors
32        for edge in current_node.edges:
33
34            # What node is at the other end of this edge?
35            v = edge.other_end(current_node)
36
37            # Visited nodes are already minimized, skip them
38            if v not in unvisited:
39                continue
40
41            # Is this a shorter route?
42            alt = dist[current_node] + edge.cost
43            if alt < dist[v]:
44
45                # Update the distance and prev dicts
46                dist[v] = alt
47                prev[v] = current_node
48
49    return (dist, prev)

```

Each time we visit a node, we have a decision to make:

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

This action, the core foundation of Dijkstra's, is the last part of the loop above. In Dijkstra's, the node with the minimum tentative distance can also be finalized, provided all edge

weights are non-negative. Nodes left at infinity are not reachable from the chosen root node.

Append some code to your `cities.py` that tests this method:

```

1 (cost_from_long_beach, prev) = network.cost_from_node(long_beach)
2 print(f"\nMinimum costs from Long Beach = {cost_from_long_beach}")
3 print(f"\nLast city before = {prev}")
4
5 nyc_cost = cost_from_long_beach[nyc]
6
7 if nyc_cost < math.inf:
8     print(f"\n*** Total cost from Long Beach to NYC: ${nyc_cost:.2f} ***")
9 else:
10    print("You can't get to NYC from Long Beach")

```

When you run it, you should get a list of how much it costs to ship a container to each city from Long Beach:

```

1 Minimum costs from Long Beach = {(node:Long Beach, edges:1): 0.0,
2 (node:Los Angeles, edges:3): 12.0, (node:Denver, edges:3): 35.0,
3 (node:Phoenix, edges:3): 31.0, (node:Louisville, edges:3): 49.0,
4 (node:Cleveland, edges:4): 44.0, (node:Boston, edges:2): 57.0,
5 (node:New York City, edges:3): 52.0}

```

You will also get a collection of node pairs. What are these? For each node, you get the node that you would pass through on the cheapest route from Long Beach:

```

1 Last city before = {(node:Los Angeles, edges:3):(node:Long Beach, edges:1),
2 (node:Denver, edges:3):(node:Los Angeles, edges:3),
3 (node:Phoenix, edges:3):(node:Los Angeles, edges:3),
4 (node:Louisville, edges:3):(node:Denver, edges:3),
5 (node:Cleveland, edges:4):(node:Denver, edges:3),
6 (node:New York City, edges:3):(node:Cleveland, edges:4),
7 (node:Boston, edges:2):(node:Cleveland, edges:4)}

```

Your users will not want to read this; give them the shortest path as a list. Add a function to `graph.py` that turns the `prev` table into a path of nodes that lead from the origin to the destination:

```

1 def shortest_path(prev, destination):
2
3     # Include the destination in the path
4     path = [destination]
5     current_node = destination

```

```

6
7     # Keep stepping backward in the path
8     while current_node in prev:
9
10         # What node should come before the current node?
11         previous_node = prev[current_node]
12
13         # Insert it at the start of the list
14         path.insert(0, previous_node)
15         current_node = previous_node
16
17     return path

```

Test that out:

```

1 if nyc_cost < math.inf:
2     print(f"*** Total cost from Long Beach to NYC: ${nyc_cost:.2f} ***")
3
4     path_to_nyc = graph.shortest_path(prev, nyc)
5     print(f"*** Cheapest path from Long Beach to NYC: {path_to_nyc} ***")
6 else:
7     print("You can't get to NYC from Long Beach")

```

This should look like this:

```

1 *** Total cost from Long Beach to NYC: $52.00 ***
2 *** Cheapest path from Long Beach to NYC: [(node:Long Beach, edges:1),
3 (node:Los Angeles, edges:3), (node:Denver, edges:3), (node:Cleveland, edges:4),
4 (node:New York City, edges:3)] ***

```

5.3 Bellman-Ford Algorithm

The Bellman-Ford Algorithm is another shortest path algorithm like Dijkstra's. However, unlike Dijkstra's Algorithm, Bellman-Ford can handle graphs with negative weight edges. (Both algorithms are at work right now on the internet: the How the Internet Works chapter shows how routers use them to choose paths for your packets.)

The algorithm works as follows:

1. Assign a tentative distance value for every node: set it to zero for our initial node and to infinity for all other nodes.
2. For each edge (u, v) with weight w , if the current distance to v is greater than the distance to u plus w , update the distance to v to be the distance to u plus w .

3. Repeat the previous step $|V| - 1$ times, where $|V|$ is the number of vertices in the graph.
4. After the above steps, if you can still find a shorter path, there exists a negative cycle.

If the graph does not contain a negative cycle reachable from the source, the shortest paths are well-defined, and Bellman-Ford will correctly calculate them. If a negative cycle is reachable, no solution exists (you could loop forever, getting “cheaper” each time), but Bellman-Ford will detect it.

Why would an edge ever be negative? Think of costs that can be refunds: a delivery route that earns a rebate, or a currency trade that gains money. Bellman-Ford is slower than Dijkstra’s, about $|V| \cdot |E|$ steps, so use it only when you need it.

Exercise 17 When Dijkstra Fails

A directed graph has edges $A \rightarrow B$ costing 1, $A \rightarrow C$ costing 3, $C \rightarrow B$ costing -3 , and $B \rightarrow D$ costing 1.

1. Run Dijkstra’s algorithm from A . What cost does it give for D ?
2. What is the true cheapest cost to D ? What went wrong?

Working Space

Answer on Page 59

Priority Queues

An emergency room does not treat patients in the order they arrive. A broken arm waits; a heart attack goes straight in. That is a *priority queue*: every item has a priority, and the most urgent one always comes out first.

Your Dijkstra's Algorithm from the last chapter needs exactly this: at each step, it pulls out the unvisited node with the lowest cost. In this chapter, you will build a priority queue and use it to speed Dijkstra up. Using it will look like this:

```
1 import kqueue
2
3 myqueue = kqueue.PriorityQueue()
4 myqueue.add(long_beach, 0) # Inserts first city and its cost
5 myqueue.add(san_diego, 14) # Puts San Diego after Long Beach
6 myqueue.add(los_angeles, 12) # Inserts LA between Long Beach and San Diego
7 current_city = myqueue.pop() # Returns (0, long_beach), the pair with the lowest
   ↪ cost, and removes it
```

Now, if an item gets a new priority, we need to remove it and reinsert it in the new spot.

```
1 myqueue.add(city_a, 16) # Puts it last in the queue
2 myqueue.update(city_a, 16, 13) # Moves it to between LA and San Diego
```

6.1 A Naive Implementation of the Priority Queue

Create a file called `kqueue.py`. Let us do a simple implementation that stores the priority and the data as tuple. And we will keep it sorted by the priority. If two tuples have the same priority, we will sort by the data.

Type this in to `kqueue.py`:

```
1 class PriorityQueue:
2     def __init__(self):
3         self.list = []
4
5     # Return and remove the first item
6     def pop(self):
7         if len(self.list) > 0:
```

```

8         return self.list.pop(0)
9     else:
10         return None
11
12     def __len__(self):
13         return len(self.list)
14
15     def update(self, value, old_priority, new_priority):
16         old_pair = (old_priority, value)
17         self.list.remove(old_pair)
18         self.add(value, new_priority)
19
20     def add(self, value, priority):
21         pair = (priority, value)
22         # Add it at the end
23         self.list.append(pair)
24         # Resort the list
25         self.list.sort()

```

This will work fine, but it could be much more efficient:

- Every time we add a single element, we resort the whole list.
- The function `remove` is searching the list sequentially for the item to delete.

In a minute, we will revisit these inefficiencies and make them better.

6.2 Using the Priority Queue

We are going to change `graph.py` to use the priority queue. While we are doing so, let us also shrink the memory footprint of our program a bit.

Notice that as the algorithm is running, each node is in one of three states:

- Unseen: In the earlier implementation, these were the nodes with `math.inf` as their cost.
- Seen, but not finalized: These are “unvisited”, but do not have `math.inf` as their cost.
- Finalized: These are the “visited” nodes — we know that their cost will not decrease any more.

We can shrink the memory foot print by not putting the unseen into the `dist` dictionary at all. And instead of a separate set for “unvisited”, what if we moved finalized nodes and their distances into a separate dictionary?

Rewrite the search loop in the `shortest-path` function in `graph.py`:

```

1         # Visited nodes are already minimized, skip them
2         if v in finalized_dist:
3             continue
4
5         # What is the cost to this neighbor?
6         alt = current_node_cost + edge.cost
7
8         # Is this the first time I am seeing the node?
9         if v not in seen_dist:
10
11             # Insert into the seen_dict, prev, and priority queue
12             seen_dist[v] = alt
13             prev[v] = current_node
14             pqueue.add(v, alt)
15
16         else: # v has been seen. Is this a cheaper route?
17             old_dist = seen_dist[v]
18             if alt < old_dist:
19                 # Update the seen_dict, prev, and priority queue
20                 seen_dist[v] = alt
21                 prev[v] = current_node
22                 pqueue.update(v, old_dist, alt)
23
24     return (finalized_dist, prev)

```

This should have exactly the same result, except for the unreachable nodes. If you have a graph that is not connected, there will be nodes that cannot be reached from the origin. In the old version, these had a cost of `math.inf`. Now, they will just not be in the dictionary at all. So, change `cities.py` to deal with this:

```

1     if nyc in cost_from_long_beach:
2         nyc_cost = cost_from_long_beach[nyc]
3         print(f"\n*** Total cost from Long Beach to NYC: ${nyc_cost:.2f} ***")
4
5         path_to_nyc = graph.shortest_path(prev, nyc)
6         print(f"\n*** Cheapest path from Long Beach to NYC: {path_to_nyc} ***")
7     else:
8         print("You can't get to NYC from Long Beach")

```

If you run `cities.py` now, it should behave exactly like the old version.

However, there is a bug. It will rear its head if two cities with the same cost are in the priority queue together. Change `cities.py` so that Denver and Phoenix have the same cost:

```
1 graph.WeightedEdge(12, long_beach, los_angeles)
2 graph.WeightedEdge(19, los_angeles, denver)
3 graph.WeightedEdge(19, los_angeles, phoenix)
```

Now try running it. You should get an error:

```
1 TypeError: '<' not supported between instances of 'Node' and 'Node'
```

What happened? When `add` sorts the list, Python compares tuples made up of a float and a `Node`. The float comes first in the tuple, so that is compared first. However, if the two tuples have the same priority, it then compares nodes.

The error statement says, “Nodes do not have a less-than method; I do not know how to compare them.”

Each `Node` lives at an address in memory. You can get that address as a number using the `id` function. The ID is unique and constant over the life of the object. It is a rather arbitrary ordering, but it will work for this problem. Add a method to your `Node` class:

```
1 # Nodes will be ordered by their location in memory
2 def __lt__(self, other):
3     return id(self) < id(other)
```

Fixed.

Now, let us make the priority queue more efficient.

6.3 Binary Search

The phone company in every town used to print a thing called a phone book. The names and phone numbers were arranged alphabetically. As you might imagine, these books often had more than a thousand pages.

If you were looking for “John Jeffers”, you would not start at the first page and read sequentially until you reached his name. You would open the book in the middle, and see a name like “Mac Miller”, then think “Jeffers comes before Miller”. You would then split the pages in your left hand in half and see a name like “Hester Hamburg” and think “Jeffers comes after Hamburg”. You would then split the pages in your right hand, and continue on like this until you found the page with “John Jeffers” on it. That is binary search. Each step cuts the search range in half, so searching a list of n items takes about

$\log_2 n$ steps. A list of 1000 items needs about 10 steps, instead of up to 1000.

Binary Search is a search algorithm that finds the position of a target value within a sorted array. The binary search algorithm works by repeatedly dividing the search interval in half. If the target value is equal to the middle element of the array, the position is returned. If the target value is less than or greater than the middle element, the search continues in the lower or upper half of the array, respectively.

6.4 Algorithm

The binary search algorithm can be described as follows:

1. If the array is empty, the search is unsuccessful, so return "Not Found".
2. Otherwise, compare the target value to the middle element of the array.
3. If the target value matches the middle element, return the middle index.
4. If the target value is less than the middle element, repeat the search with the lower half of the array.
5. If the target value is greater than the middle element, repeat the search with the upper half of the array.
6. Repeat steps 2-5 until the target value is found or the array is exhausted.

```

1  class PriorityQueue:
2      def __init__(self):
3          self.list = []
4
5      # Return and remove the first item
6      def pop(self):
7          if len(self.list) > 0:
8              return self.list.pop(0)
9          else:
10             return None
11
12     def __len__(self):
13         return len(self.list)
14
15     def add(self, value, priority):
16         pair = (priority, value)
17         i = self.loc_for_pair(pair)
18         self.list.insert(i, pair)
19
20     def update(self, value, old_priority, new_priority):
21         old_pair = (old_priority, value)
22         i = self.loc_for_pair(old_pair)
23         del self.list[i]

```

```

24         self.add(value, new_priority)
25
26     def loc_for_pair(self, pair):
27         # The range where it could be is [lower, upper)
28         # Start with the whole list
29         lower = 0
30         upper = len(self.list)
31
32         while upper > lower:
33             next_split = (upper + lower) // 2
34             v = self.list[next_split]
35             if pair < v: # pair is to the left
36                 upper = next_split
37             elif pair > v: # pair is to the right
38                 lower = next_split + 1
39             else: # Found pair!
40                 return next_split
41         return lower

```

Worked example: Use binary search to find 56 in the sorted list [2, 5, 8, 12, 16, 23, 38, 56, 72, 91], which has indices 0 through 9.

Solution: Start with `lower = 0` and `upper = 10`.

1. `next_split = 5`, and the value there is 23. Since $56 > 23$, set `lower = 6`.
2. `next_split = 8`, and the value there is 72. Since $56 < 72$, set `upper = 8`.
3. `next_split = 7`, and the value there is 56. Found it at index 7.

Exercise 18 Binary Search Steps

Use the same list, [2, 5, 8, 12, 16, 23, 38, 56, 72, 91], and the same method. Find 38. List each value that the search checks, and give the index of 38.

Working Space

Answer on Page 59

Run `cities.py` again. It should give exactly the same answers as before, only faster.

You now have a graph class that will find the cheapest path quickly, even if it has thousands of nodes with thousands of edges.

Dynamic Programming

Type “library” into your phone, and it suggests “library.” Run `git diff` after editing your essay, and Git shows exactly which lines you added and which you deleted. Biologists line up the DNA of a human and a chimpanzee to see which genes they share. All three are the same problem: comparing two sequences to find what they have in common.

The obvious way to solve it, trying every possibility, would take longer than the age of the universe for an essay of any real length. The trick that makes it fast is called *dynamic programming*: break a problem into smaller pieces that overlap, solve each piece *once*, and write the answers in a table so you never solve it again. In this chapter, you will use it to find the longest common subsequence of two strings, and then turn that into a working diff tool.

7.1 Remember Fibonacci?

In the Recursion chapter, you saw that the obvious recursive Fibonacci function is painfully slow, because `fib(n - 1)` and `fib(n - 2)` recompute the same smaller values over and over. You fixed it with *memoization*: save each answer in a dictionary the first time you compute it, and look it up after that.

There is a second way to get the same speedup. Instead of starting at the top and recursing down, start at the bottom and fill in a table, smallest values first:

```
1 def fib(n):
2     table = [0, 1]
3     for k in range(2, n + 1):
4         table.append(table[k - 1] + table[k - 2])
5     return table[n]
6
7 print(fib(80))    # 23416728348467685
```

By the time the loop needs `table[k - 1]` and `table[k - 2]`, they are already filled in. No recursion, no repeated work: just 79 additions.

These are the two styles of dynamic programming. Memoization is *top-down*: recurse as usual, but remember answers. Filling a table is *bottom-up*: solve the small problems first, in an order that guarantees each answer is ready when it is needed.

Dynamic programming works whenever a problem has two features:

- **Optimal substructure:** the answer to the big problem can be built from answers to smaller versions of the same problem.
- **Overlapping subproblems:** the same smaller problems come up again and again. (If they did not overlap, plain recursion would already be fast.)

7.2 Longest Common Subsequence

A *subsequence* of a string is what is left after you cross out some of its letters, without changing the order of the ones that remain. Crossing out letters from STUDENT can give DENT, SUN, or STENT, but not NUTS, because that changes the order.

A *common subsequence* of two strings is a subsequence of both. The *longest common subsequence* (LCS) is the longest one there is. For STUDENT and DENTIST, it is DENT:

S T U D E N T D E N T I S T

How could you find it? You could list every subsequence of the first string and check whether each one is a subsequence of the second. But each letter is either kept or crossed out, so a string of n letters has 2^n subsequences. For a 20-letter string, that is over a million. For a 1,000-line essay compared line by line, 2^{1000} is a number with 302 digits. Brute force is hopeless.

7.2.1 Thinking Recursively

Look at just the *last* letters of the two strings. There are two cases.

The last letters match. Then that letter can end the LCS. Chop it off both strings, find the LCS of what is left, and add 1. For example, the LCS of SHOE and HOSE is 1 more than the LCS of SHO and HOS, because both end in E.

The last letters do not match. Then at least one of them is not in the LCS. We do not know which, so try both: drop the last letter of the first string, or drop the last letter of the second string, and keep whichever gives the longer answer.

If either string is empty, the LCS has length 0. That is the base case. Here it is in Python, in a file called `lcs_recursive.py`:

```
1 calls = 0
2
```

```

3 def lcs_length(a, b):
4     global calls
5     calls += 1
6     if a == "" or b == "":
7         return 0
8     if a[-1] == b[-1]:
9         return lcs_length(a[:-1], b[:-1]) + 1
10    return max(lcs_length(a[:-1], b), lcs_length(a, b[:-1]))
11
12 print(lcs_length("SHOE", "HOSE"), "after", calls, "calls")
13 calls = 0
14 print(lcs_length("DYNAMICPROG", "PROGRAMMING"), "after", calls, "calls")

```

When you run it, you should see:

```

3 after 16 calls
4 after 212591 calls

```

The answers are right, but two 11-letter words took over 200,000 calls. When the last letters do not match, the function calls itself *twice*, and those two calls ask many of the same smaller questions. Just like Fibonacci, the work grows exponentially. And just like Fibonacci, the fix is to stop recomputing.

7.2.2 Filling the Table

Every smaller question has the same form: what is the LCS of the first i letters of a and the first j letters of b ? Call the answer $L[i][j]$. The recursive reasoning becomes a rule for filling a table:

The LCS Recurrence

$$L[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1][j-1] + 1 & \text{if the } i\text{th letter of } a \text{ matches the } j\text{th letter of } b \\ \max(L[i-1][j], L[i][j-1]) & \text{otherwise} \end{cases}$$

Each cell needs only the cell above it, the cell to its left, and the cell diagonally up and to the left. So if you fill the table row by row, left to right, every cell you need is already done.

Example: Fill in the table for $a = \text{SHOE}$ (down the side) and $b = \text{HOSE}$ (across the top).

Solution: The first row and column are all 0, because one of the strings is empty. Then

fill in each row. In row S, the only match is with the S in HOSE, so that cell is $0 + 1 = 1$, and the 1 is carried to the right. In row H, the H matches the first column, giving 1. In row O, the O matches the second column: the diagonal cell is 1, so this cell is 2. In row E, the E matches the last column, and the diagonal cell is 2, so the last cell is 3.

	H	O	S	E
S	0	0	0	0
H	0	1	1	1
O	0	1	2	2
E	0	1	2	3

The bottom-right cell says the LCS has length 3.

To find the letters themselves, walk backward from the bottom-right corner. If the letters for this cell match, that letter is part of the LCS: record it and step diagonally. If not, step toward whichever neighbor (up or left) holds the larger number. The bold cells are where the walk finds matches: H, O, and E, so the LCS is HOE. This is the same idea as following the prev links backward in Dijkstra's algorithm.

7.2.3 The Code

Create a file called `lcs.py`:

```

1 def lcs_table(a, b):
2     # table[i][j] = length of the LCS of a[:i] and b[:j]
3     table = [[0] * (len(b) + 1) for i in range(len(a) + 1)]
4     for i in range(1, len(a) + 1):
5         for j in range(1, len(b) + 1):
6             if a[i - 1] == b[j - 1]:
7                 table[i][j] = table[i - 1][j - 1] + 1
8             else:
9                 table[i][j] = max(table[i - 1][j], table[i][j - 1])
10    return table
11
12 def lcs(a, b):
13     table = lcs_table(a, b)
14     # Walk back from the bottom-right corner to recover the letters
15     i, j = len(a), len(b)
16     letters = []
17     while i > 0 and j > 0:
18         if a[i - 1] == b[j - 1]:
19             letters.append(a[i - 1])
20             i -= 1
21             j -= 1
22         elif table[i - 1][j] >= table[i][j - 1]:

```

```

23         i -= 1
24     else:
25         j -= 1
26     letters.reverse()
27     return "".join(letters)
28
29 if __name__ == "__main__":
30     print(lcs("SHOE", "HOSE"))
31     print(lcs("STUDENT", "DENTIST"))
32     print(lcs("DYNAMICPROG", "PROGRAMMING"))

```

The i th letter of a is $a[i - 1]$, because Python counts from 0 while the table's row 0 stands for the empty string. The `if __name__ == "__main__":` line, which you saw in the Testing and Debugging chapter, means “run these lines only when this file is run directly.” That lets another program import `lcs_table` without printing anything.

When you run it, you should see:

```

HOE
DENT
AMIG

```

The pair that needed 212,591 recursive calls now takes a 12×12 table: 144 cells. (There can be more than one longest common subsequence; AMIG is one of several of length 4.)

How fast is it in general? The table has $(n + 1)(m + 1)$ cells, and each takes a few steps, so the time grows like nm . On a laptop, comparing two random DNA strands of 2,000 letters each (4 million cells) takes a fraction of a second. Exponential has become polynomial.

7.3 Building a Diff

In the Version Control with Git chapter, Git kept every version of your files. Run `git diff` in any repository after editing a file, and Git shows what changed since your last commit. It does this by finding the longest common subsequence of the two versions, using whole *lines* as the “letters.” Lines in the LCS are unchanged. Lines that are only in the old version were deleted, and lines that are only in the new version were added.

Since Python compares lists of lines with `==` just as it compares letters, `lcs_table` works on lists without any changes. Create `diff.py` in the same folder:

```

1 from lcs import lcs_table
2
3 def diff(old, new):
4     table = lcs_table(old, new)
5     i, j = len(old), len(new)

```

```

6     lines = []
7     while i > 0 or j > 0:
8         if i > 0 and j > 0 and old[i - 1] == new[j - 1]:
9             lines.append(" " + old[i - 1])      # in both: unchanged
10            i -= 1
11            j -= 1
12        elif j > 0 and (i == 0 or table[i][j - 1] >= table[i - 1][j]):
13            lines.append("+ " + new[j - 1])      # only in new: added
14            j -= 1
15        else:
16            lines.append("- " + old[i - 1])      # only in old: deleted
17            i -= 1
18    lines.reverse()
19    return lines
20
21    monday = ["Buy poster board", "Finish lab report", "Email Ms. Rivera",
22             "Practice trumpet", "Return library book"]
23    tuesday = ["Buy poster board", "Finish lab report", "Practice trumpet",
24             "Study for chem quiz", "Return library book"]
25
26    for line in diff(monday, tuesday):
27        print(line)

```

The walk is the same as in lcs, except that it keeps going until *both* lists are used up, and every step it takes becomes a line of output. When you run it, you should see:

```

    Buy poster board
    Finish lab report
-   Email Ms. Rivera
    Practice trumpet
+   Study for chem quiz
    Return library book

```

That is the same format Git uses. Real diff tools add a few refinements, but the heart of every one of them is this table.

7.4 The Dynamic Programming Recipe

Most dynamic programming solutions are built in the same five steps:

1. **Define the subproblem.** Say in words what one cell of the table means. (“ $L[i][j]$ is the length of the LCS of the first i letters of a and the first j letters of b .”)
2. **Write the recurrence.** Express each cell in terms of smaller cells.
3. **Set the base cases.** Fill in the cells that need no smaller ones, such as the empty strings.
4. **Choose a filling order.** Make sure every cell’s ingredients are ready before you need

them.

5. **Read off the answer**, and walk back through the table if you need more than a number.

You have met dynamic programming before without its name. The Bellman-Ford algorithm in the Finding Shortest Paths chapter is dynamic programming: after round k , each distance is the shortest path using at most k edges, built from the round before. The Viterbi algorithm, mentioned in the Introduction to Markov Chains chapter, is dynamic programming too.

The name, by the way, is not very descriptive. Richard Bellman (the same Bellman) chose it in the 1950s, when the government officials who funded his work distrusted mathematical research. He later wrote that he picked “dynamic” partly because it sounded impressive and could not be used as an insult. “Programming” meant planning, as in a TV program schedule, not writing code.

Exercise 19 By Hand

Fill in the LCS table for CRATE (down the side) and TRACE (across the top). What is the length of the LCS? Walk back through your table to find one.

Working Space

Answer on Page 59

Exercise 20 Climbing Stairs

You can climb a staircase 1 or 2 steps at a time. How many different ways are there to climb 10 steps? (Hint: your last move was either 1 step or 2 steps. Let $W[n]$ be the number of ways to climb n steps.)

Working Space

Answer on Page 60

Exercise 21 Making Change

You need to make 30 cents using pennies (1), dimes (10), and quarters (25), with as few coins as possible.

1. The *greedy* method always takes the biggest coin that fits. How many coins does it use?
2. What is the true minimum?
3. Write a function `fewest_coins(amount, coins)` that uses a table `best`, where `best[t]` is the fewest coins that make `t` cents.

Working Space

Answer on Page 60

Exercise 22 Spell Check

Suppose the only way to fix a typo is to delete letters and insert letters. Explain why the fewest changes needed to turn word `a` into word `b` is $\text{len}(a) + \text{len}(b) - 2 \cdot \text{LCS}$. Use it to find the number of changes from “library” to “library” and from “recieve” to “receive.”

Working Space

Answer on Page 60

Exercise 23 How Big Is the Table?

You compare two essays of 1,000 lines each, line by line. How many cells does the LCS table have? How many subse-

Working Space

quences does brute force have to try?

Answer on Page 60

Exercise 24 **Substring or Subsequence?**

A *substring* must be made of letters that are next to each other, like DENT in STUDENT. What is the longest common substring of SNOWBOARD and SKATEBOARD? What is their longest common subsequence?

Working Space

Answer on Page 61

Answers to Exercises

Answer to Exercise 1 (on page 4)

$12 \bmod 5 = 2$, $7 \bmod 5 = 2$, $17 \bmod 5 = 2$, and $3 \bmod 5 = 3$. The buckets are:

- Bucket 0: empty
- Bucket 1: empty
- Bucket 2: 12, 7, 17 (a chain of three keys)
- Bucket 3: 3
- Bucket 4: empty

Three keys collide in bucket 2. Looking up 17 means checking the chain in bucket 2 until we reach it.

Answer to Exercise 2 (on page 4)

- Insert 12: $h = 2$, and slot 2 is empty. Store 12 in slot 2.
- Insert 7: $h = 2$, which is taken. Try slot 3, which is empty. Store 7 in slot 3.
- Insert 17: $h = 2$, slot 2 is taken, slot 3 is taken, and slot 4 is empty. Store 17 in slot 4.
- Insert 3: $h = 3$, slot 3 is taken, slot 4 is taken, and slot 0 is empty (wrapping around). Store 3 in slot 0.

The final table is: slot 0 holds 3, slot 1 is empty, slot 2 holds 12, slot 3 holds 7, and slot 4 holds 17.

Answer to Exercise 3 (on page 6)

Every key is in the same chain, so the lookup must check each of the n keys in turn. The last key is found only after n checks, which is $O(n)$. A good hash function spreads the keys across many buckets, so each chain stays short.

Answer to Exercise 4 (on page 10)

Calls: `factorial(3)`, `factorial(2)`, `factorial(1)`, `factorial(0)`. Returns: 1, then $1 \cdot 1 = 1$, then $2 \cdot 1 = 2$, then $3 \cdot 2 = 6$.

Answer to Exercise 5 (on page 10)

```
1 def total(items):
2     if len(items) == 0:
3         return 0
4     return items[0] + total(items[1:])
```

The base case is the empty list, whose sum is 0.

Answer to Exercise 6 (on page 10)

$F(6) = 8$, so the formula gives $2 \cdot 8 - 1 = 15$ calls. Counting: `fib(5)` calls `fib(4)` and `fib(3)`, and so on. The total is 15, which matches the formula.

Answer to Exercise 7 (on page 10)

The value -1 never reaches the base case $n = 0$. Each call goes to -2 , -3 , and so on, until Python raises a `RecursionError`.

Answer to Exercise 8 (on page 16)

kiwi comes before mango, so go left to apple. kiwi comes after apple, so go right to banana. kiwi comes after banana, and banana has no right child. So kiwi becomes the right child of banana.

Answer to Exercise 9 (on page 17)

The path to `data.csv` is

`./Documents/Projects/project2/data.csv`

The depth is the number of edges from the root to the node.

In this case, it is the number of subdirectories needed to reach `data.csv`:

- (1) `./` $\rightarrow$ Documents
- (2) Documents $\rightarrow$ Projects
- (3) Projects $\rightarrow$ project2
- (4) project2 $\rightarrow$ data.csv

So `data.csv` is at a depth of 4.

Answer to Exercise 10 (on page 18)

- Level k has 2^k nodes, so the total is $1 + 2 + 4 + \dots + 2^h = 2^{h+1} - 1$. (Height 2 gives 7 nodes, like Figure 3.3.)
- Since $2^{20} \approx 1,000,000$, a balanced tree is about 20 levels tall. If the words arrive in sorted order with no balancing, the tree is a chain with height 999,999.

Answer to Exercise 11 (on page 19)

The root has two children, `t` and `i`. Under `t` are `o` (ending `to`) and `e`; under `e` are `a` (tea) and `n` (ten). Under `i` is `n` (in). That is the root plus 7 letter nodes, 8 in all. The shared prefix `te` is stored only once.

Answer to Exercise 12 (on page 27)

Algorithm: DFS (Pre-order) run recursively on a tree

```

procedure DFS(root)
  if root = null then
    return
  end if
  visit(root)
  DFS(root.left)
  DFS(root.right)
end procedure

```

▷ V
 ▷ L
 ▷ R

This algorithm defines a depth-first search on a binary tree. Because the node is visited before its subtrees, it corresponds specifically to a pre-order traversal.

Answer to Exercise 13 (on page 27)

```
procedure DFS(root)
  if root = null then
    return
  end if
  pre(root) = counter
  counter = counter + 1
  DFS(root.left)
  DFS(root.right)
  post(root) = counter
  counter = counter + 1
end procedure
```

Answer to Exercise 14 (on page 28)

Node	Pre-order Number	Post-order Number
Alan	1	14
Brad	2	7
Dean	3	4
Emily	5	6
Connie	8	13
Faith	9	10
Gordon	11	12

Answer to Exercise 15 (on page 32)

Los Angeles $\rightarrow$ Bakersfield is not chosen because it does not give the best known cost to Bakersfield. The prev pointer for Bakersfield goes to San Diego instead, since $\$21 < \$22$.

The tree that is generated is called a *predecessor tree* or *explored tree* of the resulting edges formed by only the previous edges. We can use this to track shortest paths from a given origin node.

Answer to Exercise 16 (on page 32)

Start with $A = 0$. Visit A: B is 2 and C is 5. Visit B (cost 2): through B, C becomes $2 + 1 = 3$, which is less than 5, so $C = 3$. Also $D = 2 + 6 = 8$. Visit C (cost 3): through C, D becomes $3 + 2 = 5$, which is less than 8, so $D = 5$. Visit D.

The costs are $A = 0$, $B = 2$, $C = 3$, and $D = 5$. The cheapest path to D is $A \rightarrow B \rightarrow C \rightarrow D$.

Answer to Exercise 17 (on page 37)

1. Visit A: $B = 1$, $C = 3$. Visit B (cost 1, now final): $D = 2$. Visit D (cost 2). Visit C (cost 3): the route to B through C costs $3 - 3 = 0$, but B is already finalized, so nothing changes. Dijkstra reports $D = 2$.
2. $A \rightarrow C \rightarrow B \rightarrow D$ costs $3 - 3 + 1 = 1$. Dijkstra assumed that a finalized node can never get cheaper, which is true only when every edge is nonnegative. Bellman-Ford keeps relaxing every edge and finds $B = 0$ and $D = 1$.

Answer to Exercise 18 (on page 44)

The search checks 23 (index 5), then 72 (index 8), then 56 (index 7), then 38 (index 6). The index of 38 is 6.

Answer to Exercise 19 (on page 51)

	T	R	A	C	E
	0	0	0	0	0
C	0	0	0	0	1
R	0	0	1	1	1
A	0	0	1	2	2
T	0	1	1	2	2
E	0	1	1	2	3

The length is 3. Walking back with the rules from the chapter gives RAE, and in fact RAE is the only common subsequence of length 3. (RCE does not work, because C comes before R in CRATE.)

Answer to Exercise 20 (on page 51)

The last move came from step $n - 1$ or step $n - 2$, so $W[n] = W[n - 1] + W[n - 2]$, with $W[0] = 1$ (do nothing) and $W[1] = 1$. Filling the table: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89. There are 89 ways. These are the Fibonacci numbers again.

Answer to Exercise 21 (on page 52)

1. Greedy takes a quarter, then 5 pennies: 6 coins.
2. Three dimes: 3 coins. Taking the biggest coin first is not always best.
3. For each total, try ending with each coin that fits, and keep the best:

```
1 import math
2
3 def fewest_coins(amount, coins):
4     best = [0] + [math.inf] * amount
5     for total in range(1, amount + 1):
6         for coin in coins:
7             if coin <= total and best[total - coin] + 1 < best[total]:
8                 best[total] = best[total - coin] + 1
9     return best[amount]
10
11 print(fewest_coins(30, [1, 10, 25])) # 3
```

Answer to Exercise 22 (on page 52)

Keep the letters of the LCS and change everything else: delete the $\text{len}(a) - \text{LCS}$ letters of a that are not in it, then insert the $\text{len}(b) - \text{LCS}$ letters of b that are missing. That adds up to $\text{len}(a) + \text{len}(b) - 2 \cdot \text{LCS}$, and no plan can do better, because any letters you never touch form a common subsequence.

“library” and “library” have an LCS of length 6, so the count is $6 + 7 - 12 = 1$: insert an r. “recieve” and “receive” have an LCS of length 6, so the count is $7 + 7 - 12 = 2$: delete the i and insert it after the e. A spell checker can suggest the dictionary words with the smallest counts.

Answer to Exercise 23 (on page 52)

The table has $1,001 \times 1,001 = 1,002,001$ cells, about a million, which a laptop fills in well

under a second. Brute force would have to try 2^{1000} subsequences of one essay, a number with 302 digits.

Answer to Exercise 24 (on page 53)

The longest common substring is BOARD (5 letters, side by side in both). The longest common subsequence is SBOARD (6 letters): the S at the start of each word can join BOARD, because a subsequence may skip letters.



INDEX

- Ancestor, [14](#)
- associative array, [3](#)
- AVL tree, [18](#)
- B-tree, [19](#)
- balanced, [18](#)
- balanced tree, [18](#)
- base case, [7](#)
- Bellman-Ford algorithm, [36](#)
- BFS, [21](#)
 - limitations, [29](#)
- Binary Search, [42](#)
- Binary search, [8](#)
- binary search
 - algorithm, [43](#)
 - recursive, [8](#)
- binary search tree, [14](#)
- binary tree, [14](#)
- bottom-up, [45](#)
- breadth-first search, [21](#)
- call stack, [8](#)
- chaining, [4](#)
- collisions in hash tables, [3](#)
- common subsequence, [46](#)
- depth, [14](#)
- depth-first search, [21](#)
 - recursive, [8](#)
- Descendant, [14](#)
- DFS, [21](#)
- diff, [49](#)
- Dijkstra's Algorithm, [29](#), [30](#)
- Dijkstra's algorithm, [29](#)
- Dijkstra, Edsger, [30](#)
- directory, [16](#)
- dynamic programming, [45](#)
- edges, [13](#)
- explored tree, [58](#)
- Fibonacci numbers, [9](#)
- FIFO, [22](#)
- file system
 - hierarchy, [16](#)
- folder, [16](#)
- forest, [13](#)
- frame, [8](#)
- hash function, [3](#)
 - strings, [4](#)
- hash map, [3](#)
- hash table, [3](#)
- hash tables
 - time complexity, [5](#)
- height, [14](#)
 - of a tree, [14](#)
- in-order, [21](#)
- in-order traversal, [25](#)
- iterative, [26](#)
- key-value pair, [3](#)

- leaves, 14
- left child, 14
- level, 14
- level-order, 21
- level-order traversal, 21
- linear probing, 4
- load factor, 5
- longest common subsequence, 46

- Memoization, 9
- memoization, 45

- negative cycle, 37

- open addressing, 4

- Path, 14
- post-order, 21
- post-order number, 27
- post-order traversal, 25
- pre-order, 21
- pre-order number, 27
- pre-order traversal, 24
- predecessor tree, 58
- priority queue, 39
 - binary search insertion, 42
 - in graph search, 40
 - naive implementation, 39

- queue, 22

- recursion, 7
- recursive, 7, 13, 26
- recursive case, 7
- recursive traversal, 26
- red-black tree, 18
- right child, 14
- root, 13
- rotation, 18

- shortest path, 29
- siblings, 13
- subsequence, 46
- subtree, 13
- suffix tree, 19

- top-down, 45
- traversals, 21
- tree, 13
 - tree rotation, 18
 - tree traversal, 21
 - trie, 19
 - TypeError, 42
 - weighted graph, 29