CS1010X: Programming Methodology I
Chapter 9: Lists, Sorting and Searching

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 9: Lists, Sorting and Searching

Learning Objectives

By the end of this chapter, students will be able to:

- **Manipulate** lists using indexing, slicing, and core list methods (e.g. `append`, `del`).
- **Reason** about mutation and aliasing; and **write** code that prevents unintended side effects when modifying lists.
- **Implement** a stack (LIFO) and a queue (FIFO) using Python lists, including a queue implemented with two stacks.
- **Implement** linear search for unsorted lists and binary search for sorted lists, and **state** each algorithm's worst-case time complexity.
- **Apply** binary search the answer on a given monotonic predicate over an integer range to find the minimal or maximal feasible value.
- **Perform** search, insertion, and deletion on a BST while maintaining the BST invariant, and analyze runtime in terms of tree height h.
- **Implement** DFS and BFS and apply them to tree and grid problems (e.g. flood fill) with correct visited-state handling.
- **Sort** data using `sorted()` and `.sort()` with `key` functions, **explain** stability and memory implications, and compare common sorting algorithms (bubble/selection/insertion/merge/quicksort/counting) by time and space complexity.

9.1 The Python List: Internal Memory Model

To understand why some list operations are fast ($O(1)$) and others are slow ($O(n)$), we must look at how Python stores lists in memory. A Python list is **not** a linked list. It is a **dynamic array of references**.

9.1.1 Memory Representation of a List

When you execute `lst = [10, 20]`, Python creates:

1. Two integer objects (`10` and `20`) in the heap.
2. A **list object** (the header) containing the size and a pointer to an array.
3. A contiguous **array of pointers**, where each slot points to an integer.

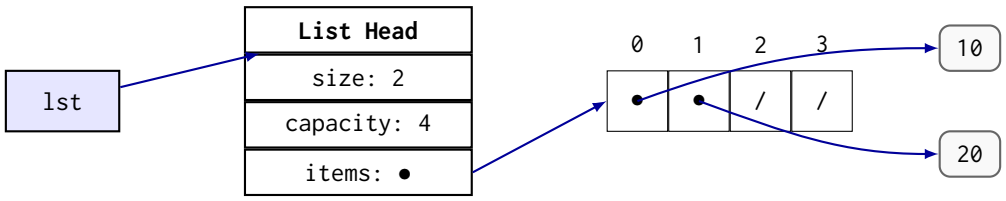


Figure 9.1: Memory model of `lst = [10, 20]`. The objects are stored separately from the array.

9.1.2 Construction and Basic Operations

The most common operations for working on lists are summarised in Table 9.1. Here n denotes the length of the list, k is the repetition factor or slice length, and m is the length of the second list.

Operation	Syntax / Example	Time	Notes
Constructors	`[]`	$O(1)$	Creates a new empty list.
	`[1, 2, 3]`	$O(n)$	Creates a list populated with literals.
	`list(iterable)`	$O(n)$	Converts iterables (e.g. tuple) into list objects.
Indexing	`lst[0]`, `lst[-1]`	$O(1)$	Random access. Negative indices count from the end. Raises `IndexError` if out of bounds.
Length	`len(lst)`	$O(1)$	Size is stored in the list header.
Concatenation	`a + b` `[1] + [2] #[1, 2]`	$O(n+m)$	Creates a **new** list object containing references from both inputs. Does not modify `a` or `b`.
Repetition	`lst * k` `[0] * 3 #[0, 0, 0]`	$O(n \cdot k)$	Returns a new list. **Warning:** The references are shallow copied (see "Aliasing" section).
Slicing	`lst[start:stop]` `lst[1:3]`	$O(k)$	Creates a **shallow copy** containing k elements. `lst[:]` is a common idiom to copy a list.

Table 9.1: Basic list operations: syntax, complexity, and behaviour.

9.1.3 List methods

Lists are ordered and *mutable*. Most methods update the list *in place* and return `None`.

Method	Example	Time	Effect
`append(x)`	`lst.append(10)`	$O(1)$ (amortised)	Add `x` at the end of the list.
`extend(it)`	`lst.extend(other)`	$O(k)$	Append all k elements from iterable `other` to the end.
`insert(i, x)`	`lst.insert(0, 'hi')`	$O(n)$	Insert `x` before position `i`, shifting later elements to the right.
`pop()` / `pop(i)`	`lst.pop()`, `lst.pop(0)`	$O(1)$ / $O(n)$	Remove and return the last element (`pop()`) or the element at index `i` (`pop(i)`). Removing from the front requires shifting elements.
`remove(x)`	`lst.remove(3)`	$O(n)$	Find and remove the first occurrence of `x`, or raise `ValueError` if not found.
`clear()`	`lst.clear()`	$O(n)$	Remove all elements from the list.
`reverse()`	`lst.reverse()`	$O(n)$	Reverse the list *in place*.
`sort()`	`lst.sort()`	$O(n \log n)$	Sort the list *in place*; uses Timsort (stable, adaptive).
`copy()`	`lst2 = lst.copy()`	$O(n)$	Return a shallow copy of the list (new list object; same element references).
`count(x)`	`lst.count(0)`	$O(n)$	Count occurrences of `x` in the list.
`index(x)`	`lst.index('a')`	$O(n)$	Return index of first `'a'`, or raise `ValueError` if not present.

9.1.4 Amortized Analysis (Not Strictly in Syllabus)

So far we have spoken about best-, worst-, and average-case *per operation* or *per input*. **Amortized analysis** is slightly different:

*We average the cost over a **sequence** of operations, not over a probability distribution of inputs.*

Formally, suppose we perform a sequence of m operations starting from an empty data structure, and let $T(m)$ be the total worst-case cost of this entire sequence. The **amortized cost per operation** is

$$\hat{c} = \frac{T(m)}{m}.$$

If $\hat{c} = O(1)$, we say the operation has *amortized constant time*, even if some individual operations may occasionally be expensive.

For example, we will see why appending to a list is $O(1)$. Consider an array-based list that:

- stores up to a certain capacity,
- when full, allocates a new array of double the size and copies all elements over (this copy takes $O(n)$ time).

A single append can therefore cost $O(n)$ in the worst case (when the resize happens).

However, over a long sequence of append operations, the expensive resizes are rare:

- When capacity is 1, we pay 1 copy;
- when capacity is 2, we pay 2 copies;
- when capacity is 4, we pay 4 copies;
- ...

If we perform m appends in total, the number of elements ever copied is

$$1 + 2 + 4 + \cdots + 2^k \;<\; 2^{k+1} \;\leq\; 2m.$$

So the total work from *all* resizes is $O(m)$. The remaining non-resizing appends each cost $O(1)$, so

$$T(m) = O(m) \quad \Rightarrow \quad \hat{c} = \frac{T(m)}{m} = O(1).$$

Thus we say that append on a list has **amortized** $O(1)$ time.

Amortized vs Average Case.

- **Average case** assumes a probability distribution over *inputs* and takes an expectation over that distribution.
- **Amortized analysis** makes *no* randomness assumptions; it guarantees that *every* sequence of m operations has total cost at most $T(m)$, so the average cost per operation in that sequence is small.

In other words, amortized bounds are worst-case guarantees *on sequences*, not on single operations.

Advanced: Amortized Analysis (The Potential Method)

The **Potential Method** is a physics-inspired technique used to determine the amortized cost of an operation sequence. It assigns a "potential energy" to the data structure.
Formal Definition: Let Φ be a potential function that maps a data structure state D to a non-negative real number $\Phi(D)$. The *amortized cost* $\hat{c}_i$ of the i-th operation is:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

where:

- c_i is the **actual cost** of the operation.
- D_i and D_{i-1} are the states after and before the operation.
- $\Phi(D_i) - \Phi(D_{i-1})$ is the change in potential ($\Delta\Phi$).

Intuition:

- If $\Delta\Phi > 0$, we overcharge the operation ($\hat{c}_i > c_i$) to store potential energy.
- If $\Delta\Phi < 0$, we undercharge ($\hat{c}_i < c_i$) by releasing stored energy to pay for an expensive operation.

We formally prove that appending to a dynamic array takes constant amortized time, even though resizing takes $O(n)$.

1. The Potential Function Let k be the number of items and n be the current capacity. We define the potential function Φ as:

$$\Phi = 2k - n$$

This function implies:

- Immediately after a resize (when $k = n/2$), $\Phi = 2(n/2) - n = 0$ (No energy).
- Just before a resize (when $k = n$), $\Phi = 2n - n = n$ (Full energy).

2. Case A: Cheap Append (No Resize)
If $k < n$, we simply add an element.

- **Actual Cost (c_i)**: 1 (write value).
- **Potential Change ($\Delta\Phi$)**: $\Phi_{new} - \Phi_{old} = (2(k+1) - n) - (2k - n) = 2$.
- **Amortized Cost ($\hat{c}_i$)**: $c_i + \Delta\Phi = 1 + 2 = \mathbf{3}$.

3. Case B: Expensive Append (Resize)
If $k = n$, the array is full. We double the capacity to $2n$, copy n items, and add the new one.

- **Actual Cost (c_i)**: n (copy) $+1$ (insert) $= n + 1$.
- **Potential Change ($\Delta\Phi$)**:

$$\begin{aligned} \Phi_{old} &= 2n - n = n \\ \Phi_{new} &= 2(n+1) - 2n = 2 \\ \Delta\Phi &= 2 - n \end{aligned}$$

- **Amortized Cost ($\hat{c}_i$)**: $(n+1) + (2-n) = \mathbf{3}$.

In both cases, the amortized cost $\hat{c}_i$ is **3**, which is a constant $O(1)$. The "expensive" resize is fully paid for by the potential energy accumulated during the "cheap" appends.

9.2 Aliasing: Lists vs. Tuples

The most critical consequence of mutability is how **aliasing** behaves. An alias occurs when two different variable names refer to the exact same object in memory.

9.2.1 Aliasing in Mutable Lists (Shared State)

Because lists are mutable, changing the object through one name affects **all** aliases.

```
a = [1, 2]
b = a           # b aliases a (same object)
b.append(3)     # Mutates the object
# Result: a is [1, 2, 3]
```

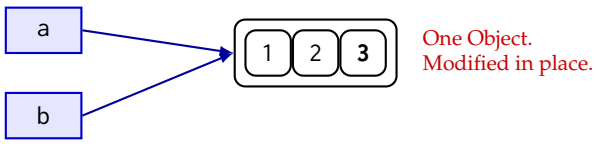


Figure 9.2: List Aliasing: a and b remain locked to the same object.

9.2.2 Comparison with Tuples (Rebinding)

Tuples are **immutable**. You cannot change them in place. Operations like += on a tuple do not mutate; they create a **new** object and rebind the variable.

```
a = (1, 2)
b = a           # b aliases a (initially same object)
b += (3,)       # Creates NEW tuple, rebinds b
# Result: a is still (1, 2)
#         b is now (1, 2, 3)
```

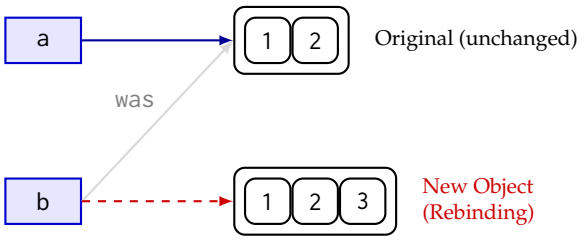


Figure 9.3: Tuple Behavior: b breaks away to point to a new object. a is unaffected.

Pitfall: Lists within Tuples

A common source of confusion is the difference between **immutable containers** and **mutable contents**.
Although a tuple is immutable, it can contain references to mutable objects (like lists).

- You **cannot** assign a new value to a tuple index (rebinding).
- You **can** modify the mutable object that the tuple points to.

```
# t is an immutable tuple containing a mutable list
t = ([10, 20], 'a')

# 1. ALLOWED: Modifying the inner list
# The tuple still points to the SAME list object, so this is
    valid.
t[0].append(99)
print(t)  # ([10, 20, 99], 'a')

# 2. FORBIDDEN: Rebinding the index
# This tries to make t[0] point to a NEW list.
t[0] = [1, 2]
# TypeError: 'tuple' object does not support item assignment
```

Key Concept: The tuple guarantees that it will always hold the *same reference* (pointer) in slot 0. It does not control what that referenced object does with its own internal data.

Understanding whether an operation creates a **new object** or modifies an **existing one** is the key to predicting the behavior of `is` and ==.

9.2.3 Summary of Operations

Operation	Syntax	Identity (`is`)	Description
Assignment	`b = a`	**Same** (`a is b`)	Creates an alias. Both names refer to the same object.
Shallow Copy	`b = a[:]`	**Different**	Creates a new container with the same contents. `a == b` is True.
Concatenation	`c = a + b`	**Different**	Always creates a completely new list object containing references from both parents.
In-Place Add	`a += b`	**Same** (for Lists)	Modifies a in place (extends it). Its identity `id(a)` does not change.
Methods	`a.append(x)`	**Same**	Mutates the object directly.

In particular, there is an important distinction between binary addition (+) and augmented assignment (+=).

The id() Function

Every object in Python has a unique identity. The built-in function `id(obj)` returns an integer representing this identity.

Key Properties:

- **Uniqueness:** No two objects with overlapping lifetimes have the same ID.
- **Consistency:** An object's ID never changes once created.
- **Under the Hood:** In the standard CPython implementation, `id(x)` returns the actual **memory address** of the object (similar to a pointer in C).

The `is` operator is essentially a readable wrapper for comparing IDs:

$$\texttt{a is b} \iff \texttt{id(a) == id(b)}$$

1. Concatenation (+) creates a new object

Using + always builds a new list. Even if you assign it to the same name, the *identity* changes.

```
a = [10, 20]
old_id = id(a)
a = a + [30]   # Creates NEW list, then rebinds 'a'
print(a)       # [10, 20, 30]
print(id(a) == old_id)  # False (a points to a new box)
```

2. In-Place Extension (+= and extend) → Same Object

For **lists**, += maps to the special method `__iadd__` (in-place add), which behaves like `extend()`.

Using +=

```
a = [10, 20]
b = a          # Alias
a += [30]      # Mutates a
print(a)       # [10, 20, 30]
print(b)       # [10, 20, 30]
print(a is b)  # True
```

Using extend()

```
a = [10, 20]
b = a          # Alias
a.extend([30]) # Mutates a
print(a)       # [10, 20, 30]
print(b)       # [10, 20, 30]
print(a is b)  # True
```

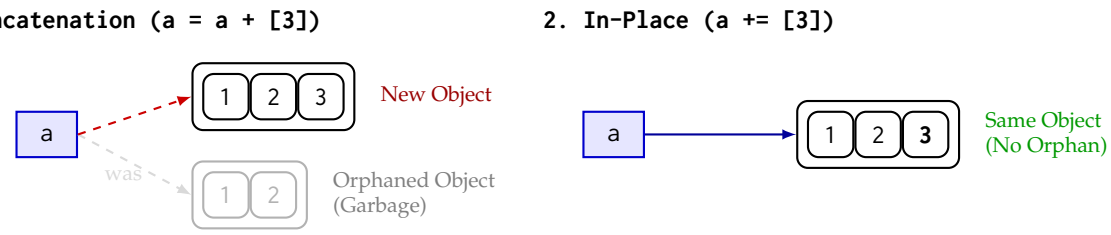


Figure 9.4: `a = a + ...` abandons the old list (creating garbage), whereas `a += ...` modifies it in-place.

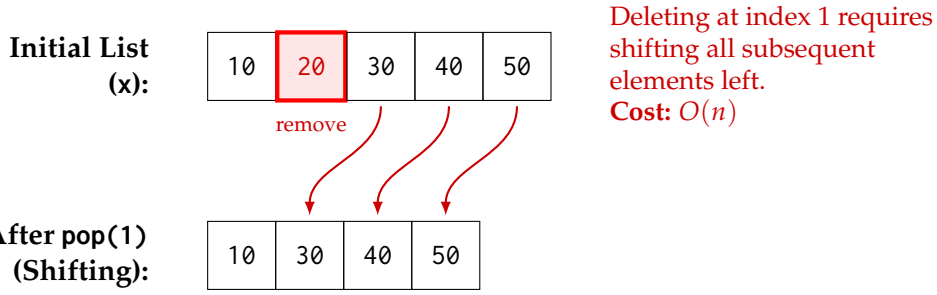
9.3 Deletion and Time Complexity

Deletion in a dynamic array is generally expensive unless done at the very end.

9.3.1 The Cost of Shifting

When you remove an item from the middle of a list (index i), all subsequent items (indices $i+1$ to $n-1$) must be **shifted left** to fill the gap.

```
x = [10, 20, 30, 40, 50]
x.pop(1)  # Removes 20 (index 1)
```



This shifting operation requires shifting $n - i$ elements. Therefore, the closer you are to the front of the list, the slower the deletion.

9.3.2 Comparison of Deletion Methods

Method	Syntax	Complexity	Mechanism
Pop (End)	`lst.pop()`	$O(1)$	Decrements size counter. No shifting required.
Pop (Index)	`lst.pop(0)`	$O(n)$	Removes item at i, shifts all subsequent items left.
Remove	`lst.remove(x)`	$O(n)$	Linear search to find x ($O(n)$) + Shift remaining items ($O(n)$).
Del (Item)	`del lst[i]`	$O(n)$	Same mechanism as `pop(i)` but returns nothing.
Del (Slice)	`del lst[i:j]`	$O(n)$	Removes a block; shifts tail left once.
Clear	`lst.clear()`	$O(n)$	Resets size to 0. (Implementation dependent, usually fast).

9.4 del Statement: Deleting Names vs. Objects

A common misconception is that `del a` deletes the object from memory. In Python, `del` removes the **variable name** from the current scope (namespace). It does not directly touch the object in the heap.

9.4.1 Reference Counting Mechanism (Out of Syllabus)

Python manages memory using *reference counting*. An object is deleted (garbage collected) only when the number of names referring to it drops to zero.

1. `del a`: Removes the name a. The reference count of the object decreases by 1.
2. If the reference count reaches 0, the object is immediately destroyed.
3. If other names (aliases) still refer to the object, the object remains alive.

```
a = [10, 20]
b = a           # Reference count = 2 (a, b)
del a           # Deletes name 'a'. Reference count = 1 (b)
                # The list [10, 20] still exists in memory!
print(b)        # [10, 20]
print(a)        # NameError: name 'a' is not defined
```

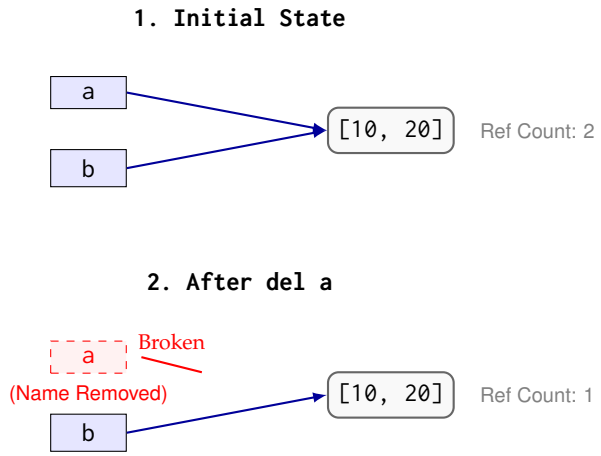


Figure 9.5: Visualizing `del a`. The name 'a' is removed, breaking the link. However, because 'b' still points to the object, the list stays in memory.

9.4.2 Comparison: `del a` vs `del a[:]`

Command	Target	Effect on Aliases
`del a`	The **Name**	**None.** Other aliases continue to see the full list object.
`del a[:]`	The **Object Contents**	**Drastic.** The object is emptied. Since aliases point to the *same object*, they all see an empty list `[]`.

9.5 Iterating Over Lists

Python provides two standard ways to traverse a list. Under the hood, both rely on the list's internal iterator.

9.5.1 Standard Traversal

The most "Pythonic" approach iterates directly over the elements.

```
nums = [10, 20, 30]
for x in nums:
    print(x)
```

If you need the index as well (e.g. to update the item), use `enumerate()`:

```
for i, x in enumerate(nums):
    print(f"Index {i} contains {x}")
```

9.5.2 The Mutation Hazard: Modifying While Iterating

A common logic error occurs when you attempt to add or remove elements from a list while currently iterating over it.

Python's internal iterator keeps track of a current **index**. If you remove the current element, all subsequent elements shift left. However, the iterator blindly moves its index to the right for the next step.

This creates a bug when two items that need removal are side-by-side.

```
# Goal: Remove ALL even numbers
nums = [1, 2, 4, 5]
for x in nums:
    if x % 2 == 0:
        nums.remove(x)

print(nums)
# Expected: [1, 5]
# Actual:   [1, 4, 5]  <-- BUG! 4 was skipped.
```

Execution Trace: To find out why `4` is not deleted:

Step	Index	List State	Action
1	0	`[1, 2, 4, 5]`	Read `nums[0]` (1). Odd. Keep.
2	1	`[1, 2, 4, 5]`	Read `nums[1]` (2). Even. **Remove 2**. *Shift left:* List is now `[1, 4, 5]`. The value `4` has shifted into index 1.
3	2	`[1, 4, 5]`	Loop increments index to 2. Read `nums[2]` (5). Odd. Keep. **Result:** Index 1 (value 4) was never checked.

9.5.3 Safe Patterns for Modification

To avoid this, we must decouple the iteration from the modification, or adjust our iteration strategy.

Pattern A: Iterating over a Copy (Decoupling)

We iterate over a snapshot (shallow copy) of the list, but perform the deletions on the original list. The copy does not change size, so indices remain stable.

```
# nums[:] creates a full slice copy
for x in nums[:]:
    if x % 2 == 0:
        nums.remove(x)  # Modifies original 'nums'
```

- **Pros:** Conceptually simple.
- **Cons:** Uses $O(n)$ extra memory for the copy. `remove` is $O(n)$, making the whole loop $O(n^2)$.

Pattern B: List Comprehension (Filter-Create)

Instead of mutating the existing object, we construct a **new** list containing only the items we wish to keep. This is the idiomatic Python approach.

```
# "Keep x if x is NOT even"
nums = [x for x in nums if x % 2 != 0]
```

- **Pros:** Fastest execution (because list comprehension is optimized in C). Easy to read.
- **Cons:** Creates a new list object (updates the reference `nums`). Does not work if other variables (aliases) need to see the changes in-place.

Pattern C: Reverse Iteration

If we iterate backwards, deleting the current element only shifts items at indices *greater* than the current index. Since we are moving towards index 0, the items we are about to visit are unaffected.

```
for i in range(len(nums) - 1, -1, -1):
    if nums[i] % 2 == 0:
        del nums[i]
```

- **Pros:** In-place modification (memory efficient). No aliases broken.
- **Cons:** Harder to read. Can be $O(n^2)$ because `del` can causes shifting of elements.

9.6 List Comprehensions

List comprehensions offer a concise way to create lists. They are syntactically derived from Set Builder Notation in mathematics, for example, $\{x^2 \mid x \in \mathbb{Z}, x > 0\}$.

9.6.1 Syntax of List Comprehension

```
[expr for var in iterable if condition]
 Map                          Filter
```

9.6.2 Relationship to Map and Filter

Any list comprehension can be rewritten using the functional `map` and `filter` primitives. Comprehensions are generally preferred for readability (and are often slightly faster in Python).

List Comprehension	Map / Filter Equivalent
Map only:	
`L = [x**2 for x in nums]`	`L = list(map(lambda x: x**2, nums))`
Filter only:	
`L = [x for x in nums if x > 0]`	`L = list(filter(lambda x: x > 0, nums))`
Combined:	
`L = [x**2 for x in nums if x > 0]`	`L = list(map(lambda x: x**2, filter(lambda x: x > 0, nums)))`

9.6.3 Nested Comprehensions

You can use multiple `for` clauses. The order matches nested loops (left-to-right is outer-to-inner).

```
# Cartesian Product
colors = ['red', 'blue']
sizes = ['S', 'M']

items = [(c, s) for c in colors for s in sizes]
# Result: [('red','S'), ('red','M'), ('blue','S'), ('blue','M')]
```

Comprehension vs Generator Expression

Note that replacing the square brackets `[]` with parentheses `()` creates a **Generator**, not a Tuple.

- `[x*2 for x in range(3)]` → List (evaluates immediately, uses memory).
- `(x*2 for x in range(3))` → Generator (lazy evaluation, $O(1)$ memory).

9.6.4 Pitfall: Initialization by Multiplication

A common mistake when initializing 2D lists (matrices) is using the repetition operator $*$ for the outer dimension.

```
n = 3
# DANGEROUS: Creates references to the SAME list!
matrix = [[0] * n] * n

matrix[0][0] = 99
# Result: [[99, 0, 0], [99, 0, 0], [99, 0, 0]]
# All rows are modified because they are the same object.
```

This happens due to **aliasing**. The expression `[[0]*n] * n` creates a single list `[0, 0, 0]`, and then creates a list containing n references to that **one specific object**.

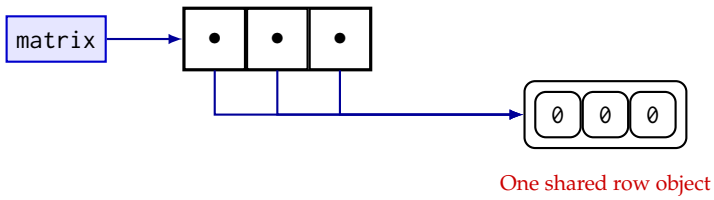


Figure 9.6: The "List of Lists" Multiplication Trap: All slots point to the same row.

Therefore, to create independent rows, we can use a list comprehension. This ensures the inner list expression `[1] * n` is **evaluated** n **times**, creating n distinct objects.

```
# CORRECT: Evaluates the inner list fresh every time
matrix = [[0] * n for _ in range(n)]

matrix[0][0] = 99
# Result: [[99, 0, 0], [0, 0, 0], [0, 0, 0]]
# Only the first row is affected.
```

9.7 Lists as Stacks and Queues

While Python lists are implemented as dynamic arrays, their performance characteristics ($O(1)$ append and pop at the end) make them excellent candidates for implementing **Stacks**. With some algorithmic ingenuity, they can also efficiently simulate **Queues**.

9.7.1 The Stack ADT (LIFO)

A **Stack** is a collection that follows the **L**ast-**I**n, **F**irst-**O**ut principle. The last element added is the first one removed.

- **Push:** Add an element to the top.
- **Pop:** Remove and return the top element.
- **Peek:** View the top element without removing it.

```
stack = []
stack.append(10)  # Push: O(1)
stack.append(20)  # Push: O(1)
top = stack.pop() # Pop:  O(1) -> Returns 20
```

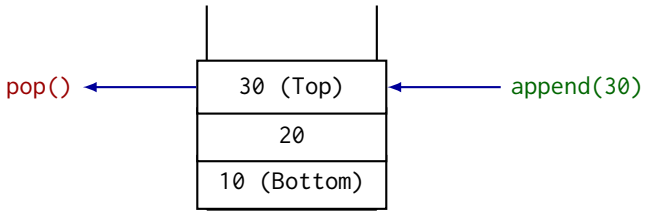


Figure 9.7: Python List as a Stack. Operations are performed at the end of the list.

9.7.2 Monotonic Stacks

A **Monotonic Stack** is a specialized usage pattern where the elements inside the stack are always maintained in a sorted order (either strictly increasing or strictly decreasing).

This data structure is the standard solution for finding the **Next Greater Element (NGE)** or **Next Smaller Element (NSE)** for every item in an array in $O(N)$ time.

There are two common variants, but they make use of the same idea:

- **Monotonic Decreasing Stack:** Elements are sorted large $\rightarrow$ small (e.g. `[10, 5, 2]`).
  - Useful for finding the **Next Greater Element**.
  - *Logic:* If we encounter a new number X larger than the stack top, the order is threatened. We must pop elements until order is restored. The element X is the "Next Greater" for every element popped.
- **Monotonic Increasing Stack:** Elements are sorted small $\rightarrow$ large.
  - Useful for finding the **Next Smaller Element**.

To understand the algorithm, we view it through the lens of its **invariant**. For the "Next Greater Element" problem, the invariant is:

"The stack always contains elements in strictly decreasing order."

When we process a new element X:

1. **Conflict:** If $X >$ `stack[-1]`, pushing X would violate the invariant.
2. **Resolution:** We must pop the top element.
3. **Deduction:** The act of popping an element Y because of X implies that X is the **first** element to the right of Y that is larger than it. Thus, X is the NGE of Y.

Algorithm and Implementation

```python
def next_greater_elements(nums):
    result = [-1] * len(nums)
    stack = []  # Stores INDICES, not values

    for i, current_val in enumerate(nums):
        # Invariant Maintenance:
        # While the new value is greater than the stack top...
        while stack and nums[stack[-1]] < current_val:
            idx = stack.pop()    # Resolve conflict
            result[idx] = current_val # Record the NGE

        stack.append(i) # Push current index to wait

    return result
```

Execution Trace

Consider the array: `nums = [2, 1, 5, 6, 2, 3]`.

Current	Action	Stack State	Reasoning
2	Push 0	`[2]`	Stack empty. 2 waits.
1	Push 1	`[2, 1]`	$1 < 2$. Invariant holds.

5	**Pop 1**
	Pop 0
	Push 2
		`[2]`
		`[]`
		`[5]`
			$5 > 1$. NGE(1) $\rightarrow$ 5.
			$5 > 2$. NGE(0) $\rightarrow$ 5.
			5 waits.

6	**Pop 2** Push 3	`[]` `[6]`	$6 > 5$. NGE(2) $\rightarrow$ 6. 6 waits.
2	Push 4	`[6, 2]`	$2 < 6$. Invariant holds.
3	**Pop 4** Push 5	`[6]` `[6, 3]`	$3 > 2$. NGE(4) $\rightarrow$ 3. $3 < 6$. 3 waits.

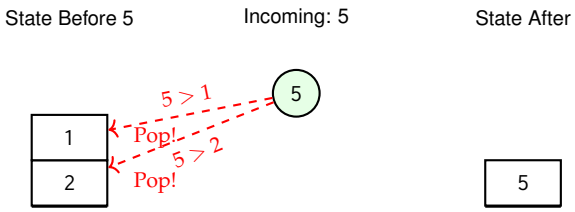


Figure 9.8: The Monotonic Invariant. When 5 arrives, it forces smaller elements (1, 2) out.

Complexity Analysis

Despite the nested `while` loop, the time complexity is **linear**.

- $O(N)$ **Time:** Each element is pushed onto the stack exactly once and popped at most once. The total operations are proportional to $2N$.
- $O(N)$ **Space:** In the worst case (strictly decreasing input), the stack holds all N elements.

9.7.3 The Queue ADT (FIFO)

A **Queue** is a collection that follows the **F**irst-**I**n, **F**irst-**O**ut (FIFO) principle. It behaves exactly like a line of people waiting for a bus: the first person to queue up is the first person to board.

Key Operations:

- **Enqueue (Push):** Add an element to the **back** (rear) of the queue.
- **Dequeue (Pop):** Remove and return the element from the **front** (head) of the queue.

While Python provides a specialized `collections.deque` for this, understanding how to implement a queue using stacks is a classic exam / interview problem.

9.7.4 Queue using Two Stacks

Implementing a Queue using two Stacks allows us to build a FIFO data structure using only LIFO operations.

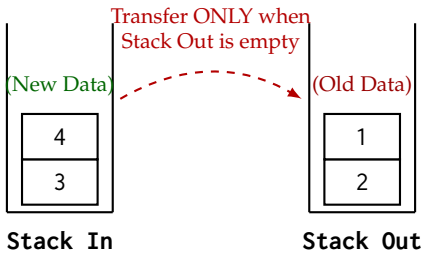
The Strategy

We utilize two lists (stacks) to separate the input and output streams:

1. `stack_in`: Handles all incoming data (**Enqueue**).
2. `stack_out`: Handles all outgoing data (**Dequeue**).

Logic:

- **To Enqueue(x):** Simply push x onto `stack_in`. This is always $O(1)$.
- **To Dequeue():** We need the "oldest" element.
  - If `stack_out` is **not empty**, the top element is the oldest. Pop it.
  - If `stack_out` is **empty**, we perform a **Transfer**. We pop *everything* from `stack_in` and push it onto `stack_out`.



1. Enqueue adds to 'Stack In'.
2. Dequeue pops from 'Stack Out'.
3. If 'Stack Out' is empty, pour In $\rightarrow$ Out.

Figure 9.9: Queue via Two Stacks. The transfer operation reverses the order, turning LIFO into FIFO.

Since a stack reverses order (LIFO), moving items from one stack to another reverses them again.

$$\text{Reverse}(\text{Reverse}(\text{Input})) \rightarrow \text{Original Order (FIFO)}$$

```python
# Global lists acting as stacks
stack_in = []
stack_out = []

def enqueue(x):
    stack_in.append(x)

def dequeue():
    if not stack_out:
        # Transfer mechanism: Lazy migration
        # Only happens when out-stack is empty!
        while stack_in:
            stack_out.append(stack_in.pop())

    if not stack_out:
        # If still empty after transfer, queue is truly empty
        raise IndexError("Dequeue from empty queue")

    return stack_out.pop()
```

Amortized Analysis

Ideally, a queue operations should be $O(1)$. In this implementation:

- **Enqueue** is always $O(1)$.
- **Dequeue** is usually $O(1)$, but occasionally $O(N)$ when a transfer occurs.

However, the **amortized cost** is $O(1)$. Consider the lifecycle of a single element X:

1. Pushed onto `stack_in` (1 operation).
2. Popped from `stack_in` during transfer (1 operation).
3. Pushed onto `stack_out` during transfer (1 operation).
4. Popped from `stack_out` during dequeue (1 operation).

Every element undergoes exactly 4 operations. Thus, for a sequence of N elements, the total work is $4N$, which averages to $O(1)$ per operation.

9.8 Searching in Lists

Searching is a fundamental operation in computer science that involves locating a specific target value (the *key*) within a collection of data. In this section, we explore two primary algorithms for searching in Python lists: **Linear Search** and **Binary Search**.

9.8.1 The Searching Problem

Problem Definition: Given a list L of n elements and a target value k, determine:

1. **Existence:** Does k exist in L?
2. **Location:** If yes, what is the index i such that $L[i] == k$? (Typically the first such index).

9.8.2 Linear Search (Sequential Search)

Linear search is the most basic searching algorithm. It requires **no assumptions** about the order of elements in the list.

The algorithm proceeds sequentially from the first element to the last:

1. Start at index 0.
2. Compare the current element with the key k.
3. If they match, return the current index (success).
4. If not, move to the next index.
5. If the end of the list is reached without a match, return `None` (failure).

```
def linear_search(key, lst):
    """
    Scans the list from start to end to find the key.
    Returns the index if found, otherwise None.
    """
    for i in range(len(lst)):
        if lst[i] == key:
            return i
    return None
```

Consider searching for the key **3** in the list `[5, 2, 3, 4]`.

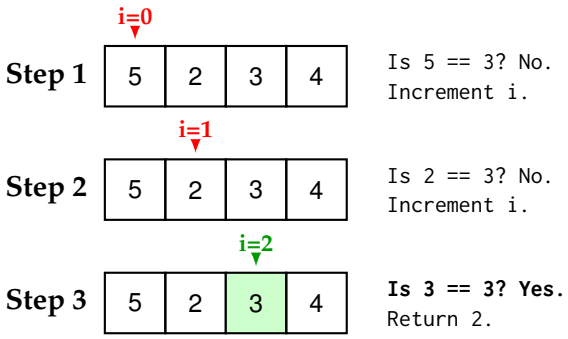


Figure 9.10: Linear Search for target 3. The algorithm scans index by index.

9.8.3 Complexity Analysis of Linear Search

Let n be the number of elements in the list.

1. Time Complexity

- **Best Case ($\Omega(1)$):** The key is found immediately at the first index (index 0). Only 1 comparison is required.
- **Worst Case ($O(n)$):** The key is either at the very last index or not present in the list at all. The algorithm must check all n elements.
- **Average Case ($\Theta(n)$):**
  - **Assumption:** The key is present in the list, and it is equally likely to be at any index from 0 to $n-1$.
  - **Derivation:** The probability of the key being at any specific index is $\frac{1}{n}$. If the key is at index i, we perform $i+1$ comparisons.

$$E[\text{comparisons}] = \sum_{i=0}^{n-1} \underbrace{(i+1)}_{\text{cost}} \cdot \underbrace{\left(\frac{1}{n}\right)}_{\text{probability}} = \frac{1}{n}\sum_{k=1}^{n} k = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

  - **Conclusion:** On average, we scan half the list. Since constants are ignored in asymptotic notation, this remains linear time $O(n)$.

2. Space Complexity

- **Space Cost ($O(1)$):** Linear search is an **in-place** algorithm. It only requires a constant amount of extra memory for a loop counter (`i`) and comparison logic, regardless of the input size n. It does not copy the list or use recursive stack frames.

9.8.4 Binary Search (Divide and Conquer)

If the list is **sorted**, by checking the middle element, we can tell if the element lies to the left or right of the current element. Therefore, we can ignore half of the remaining elements after just one comparison. This is the principle of Binary Search, a classic Divide and Conquer algorithm.

Precondition

Binary Search works ONLY on sorted lists. Applying it to an unsorted list will yield incorrect results.

We maintain two pointers, `low` and `high`, which define the current search interval $[low, high]$.

1. **Calculate Middle:** Find the middle index: `mid` $=$ (`low` $+$ `high`)`//2`.
2. **Compare:** Check the value at `lst[mid]` against the `key`:
   - **Match:** If `lst[mid] == key`, we found it! Return `mid`.
   - **Too Small:** If `key < lst[mid]`, the key cannot be in the right half. We discard the right side by updating `high = mid - 1`.
   - **Too Large:** If `key > lst[mid]`, the key cannot be in the left half. We discard the left side by updating `low = mid + 1`.
3. **Repeat:** Continue this process until the key is found or the interval becomes empty (`low` $>$ `high`).

9.8.5 Iterative Implementation of Binary Search

The iterative approach is generally preferred for production code because it is slightly faster and uses constant $O(1)$ memory.

```
def binary_search_iterative(key, lst):
    low = 0
    high = len(lst) - 1

    while low <= high:
        mid = (low + high) // 2
        guess = lst[mid]

        if guess == key:
            return mid          # Found match
        elif guess > key:
            high = mid - 1      # Discard right half
        else:
            low = mid + 1       # Discard left half

    return None                 # Not found
```

9.8.6 Recursive Implementation of Binary Search

Binary search can also be defined naturally using recursion. We pass the current bounds `low` and `high` as arguments to the recursive function.

```python
def binary_search_recursive(key, lst, low, high):
    # Base Case: Interval is empty
    if low > high:
        return None

    mid = (low + high) // 2
    guess = lst[mid]

    if guess == key:
        return mid
    elif guess > key:
        # Recurse on the left half
        return binary_search_recursive(key, lst, low, mid - 1)
    else:
        # Recurse on the right half
        return binary_search_recursive(key, lst, mid + 1, high)
```

For example, suppose we wanted to find the key **25** in `[5, 9, 12, 18, 25, 34, 85, 100]`.

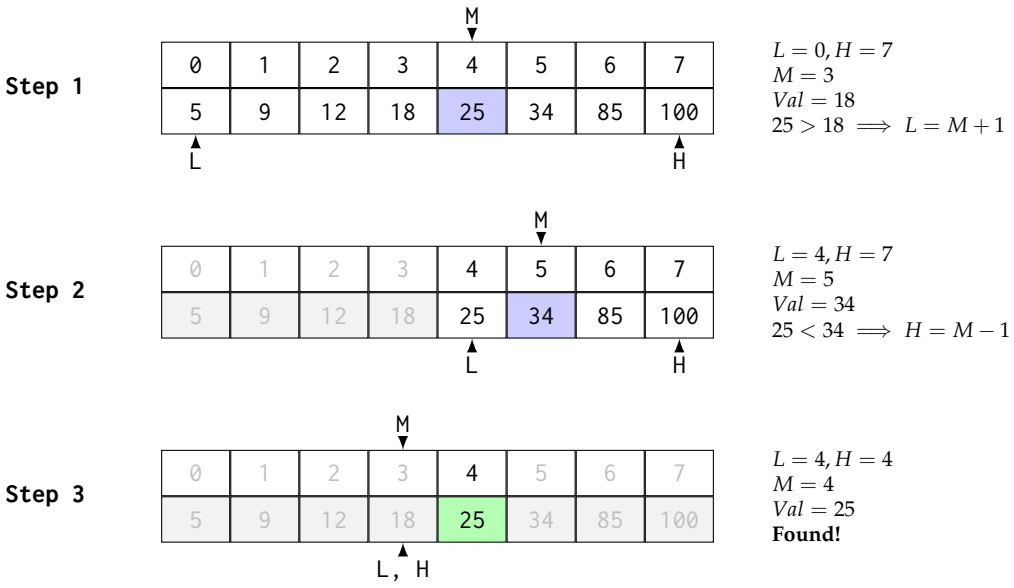


Figure 9.11: Trace of Binary Search. L and H track the range, M tracks the middle index.

9.8.7 Complexity Analysis of Binary Search

Time Complexity ($O(\log n)$)

For both the iterative and recursive implementations, the time complexity is determined by how many times we can divide the search space n by 2 before it becomes trivial (size 1). Mathematically, we look for the number of steps k such that:

$$\frac{n}{2^k} = 1 \implies n = 2^k \implies k = \log_2 n$$

- **Best Case ($O(1)$):** The target key is found at the very first comparison (the middle element of the initial array).
- **Worst Case ($O(\log n)$):** The target key is not in the list, or is found only at the deepest level of the search tree (when the interval size reduces to 1).
- **Average Case Analysis ($O(\log n)$):** We assume the list contains $n = 2^k - 1$ items and the target is chosen uniformly at random (probability $1/n$). We can group the elements by how many comparisons are needed to find them:
  1. **1 Comparison:** There is exactly **1** element (the initial middle) that requires only 1 comparison.
  2. **2 Comparisons:** There are **2** elements (the middles of the left and right halves) that require 2 comparisons.
  3. **3 Comparisons:** There are **4** elements that require 3 comparisons.
  4. d **Comparisons:** In general, there are 2^{d-1} elements that require exactly d comparisons.

  To find the **expected value** $E[C]$, we sum the probability times the cost for all groups up to k levels:

$$E[C] = \sum_{d=1}^{k} \underbrace{d}_{\text{Cost}} \times \underbrace{\frac{2^{d-1}}{n}}_{\text{Probability}} = \frac{1}{n} \sum_{d=1}^{k} d \cdot 2^{d-1}$$

  Using the summation identity $\sum_{i=1}^{k} i2^{i-1} = (k-1)2^k + 1$:

$$E[C] = \frac{1}{n}\left((k-1)2^k + 1\right)$$

  Since $n \approx 2^k$, we substitute 2^k with n:

$$E[C] \approx \frac{1}{n}\left((k-1)n\right) \approx k - 1$$

  Since $k = \log_2 n$, the average number of comparisons is $\approx \log_2 n - 1$. Thus, the average case is $\Theta(\log n)$.

Space Complexity (Memory Usage)

This is where the two implementations differ significantly.

- **Iterative:** $O(1)$. It only uses a constant amount of memory for three variables (`low`, `high`, `mid`).
- **Recursive:** $O(\log n)$. Each recursive call creates a new "stack frame" in memory. Since the depth of recursion is $\log n$ (the height of the recursion tree), it uses $O(\log n)$ stack space.

9.8.8 Advanced: Binary Search the Answer (BSTA)

Sometimes, we are not searching for an element in an explicit list. Instead, we are searching for a specific **value** (the "Answer") that satisfies a condition within a range of possibilities.

This technique is applicable when the problem has a **Monotonic Property**:

> *"If a value X is a valid solution, then all values greater than X are also valid (or invalid)."*

Instead of an index in an array, our `low` and `high` represent the **range of possible answers** (e.g. from 0 to 10^9).

1. Define the search space: `low` (minimum possible answer) and `high` (maximum possible answer).
2. Define a `check(mid)` function: Returns `True` if `mid` is a feasible solution.
3. Perform Binary Search:
    - Calculate `mid`.
    - If `check(mid)` is `True`: This is a valid solution, but we want the **minimal** one. We record `mid` as a potential answer and try to find a smaller one by moving left (`high = mid - 1`).
    - If `check(mid)` is `False`: This value is insufficient. We need a larger value, so we move right (`low = mid + 1`).

For example, consider the problem below:

You have a conveyor belt that must ship N packages within D days. The i-th package has weight w_i. What is the **minimum weight capacity** of the ship so that you can transport all packages within D days?

Monotonicity:

- If a capacity C works, any capacity $C' > C$ definitely works (monotonic).
- We want the *smallest* C that works.

Therefore, we can write the solution as:

```python
def min_capacity(weights, days):
    def check(capacity):
        # Simulation: Can we ship within 'days' with this capacity?
        current_load = 0
        days_needed = 1
        for w in weights:
            if current_load + w > capacity:
                days_needed += 1
                current_load = 0
            current_load += w
        return days_needed <= days

    # Range of possible answers
    low = max(weights)  # Must carry at least the heaviest item
    high = sum(weights) # Worst case: carry everything in 1 day
    ans = high

    while low <= high:
        mid = (low + high) // 2
        if check(mid):
            ans = mid      # mid works, try smaller
            high = mid - 1
        else:
            low = mid + 1  # mid too small, try larger

    return ans
```

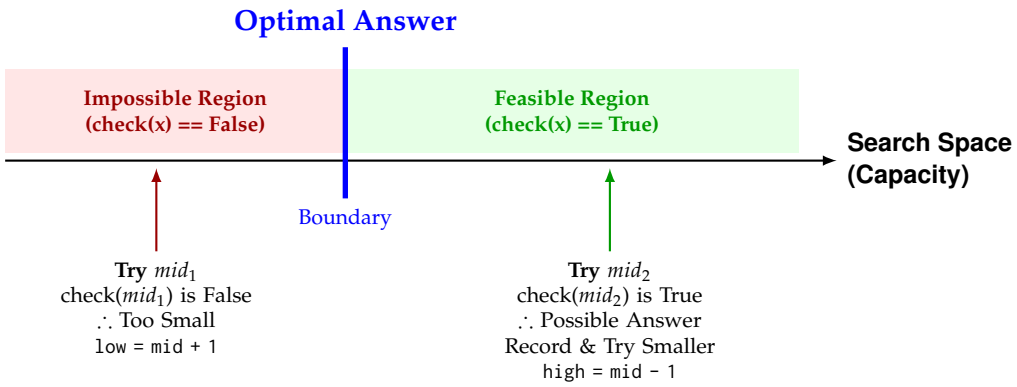


Figure 9.12: Visualizing Binary Search on the Answer . We search for the boundary where the condition switches from False to True.

9.9 Binary Search Trees (BST)

A **Binary Search Tree** is a hierarchical data structure that brings the efficiency of Binary Search ($O(\log n)$) to a dynamic tree structure. Unlike a sorted list, which is slow to update ($O(n)$ insertion/deletion), a BST supports fast updates.

9.9.1 Structure and Invariant

We can represent a BST node as a tuple: `(Left_Subtree, Value, Right_Subtree)`. An empty tree is represented by `None`.

The BST Property: For every node N with value V:

- All values in the **left subtree** are strictly **smaller** than V.
- All values in the **right subtree** are strictly **larger** than V.

Since tuples in Python are **immutable**, our operations cannot change the tree in place. Instead, they must return a **new tuple** representing the modified tree.

There are 3 key operations for working with binary search trees:

- **Searching:** Finding if an element exists in the BST in $O(\log n)$ time.
- **Inserting:** Insert an element into the BST in $O(\log n)$ time.
- **Deletion:** Find and delete an element in the BST in $O(\log n)$ time.

Because we do not rebalance our trees in this simple example, we will express the time complexities in terms of the height of the BST, h.

Terminology: Height of a Tree

In this course, we define the **height** of a tree as follows:

- **Empty Tree:** Height is 0.
- **Non-Empty Tree:** 1+ the maximum height of its subtrees.

That is, it is the number of nodes on the longest path (*Some texts use the number of edges, in which case, an empty tree has height -1 and a singular node has height 0*).

Thus, we can find the height of a tree as follows:

```
def get_height(tree):
    # Base Case: Empty tree has height 0
    if tree is None:
        return 0

    left, val, right = tree

    # Recursive Step: 1 + max height of children
    return 1 + max(get_height(left), get_height(right))
```

9.9.2 Searching in a BST ($O(h)$)

We compare the target with the current node and move strictly left or right.

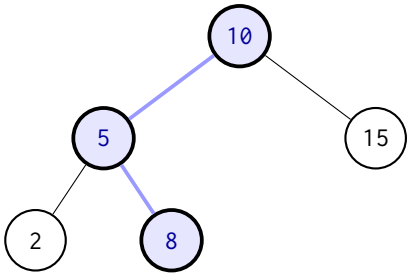
```python
def bst_search(tree, target):
    # Base Case: If the tree is empty (None), the target is not found.
    if tree is None:
        return False

    # Unpack the current node: (Left Child, Value, Right Child)
    left, val, right = tree

    # Case 1: Target matches the current node's value. Found!
    if target == val:
        return True

    # Case 2: Target is smaller than current value.
    # Recursively search the Left Subtree (ignore the right).
    if target < val:
        return bst_search(left, target)

    # Case 3: Target is larger than current value.
    # Recursively search the Right Subtree (ignore the left).
    return bst_search(right, target)
```



Target: 8
1. Start at 10. $8 < 10 \rightarrow$ Left.
2. At 5. $8 > 5 \rightarrow$ Right.
3. At 8. Found!

Figure 9.13: Visualizing BST Search. The path (highlighted) is determined solely by comparing values.

9.9.3 Insertion into a BST ($O(h)$)

To insert a value X:

1. Compare X with the current node.
2. Traverse down until you hit `None` (an empty spot).
3. Create a new leaf node there.
4. **Rebuild** the path back up (since tuples are immutable, we create new parent tuples pointing to the new children).

```python
def bst_insert(tree, x):
    if tree is None:
        return (None, x, None)

    left, val, right = tree
    if x < val:
        new_left = bst_insert(left, x)
        return (new_left, val, right) # Rebuild node
    elif x > val:
        new_right = bst_insert(right, x)
        return (left, val, new_right) # Rebuild node
    else:
        return tree # Duplicate found, do nothing
```

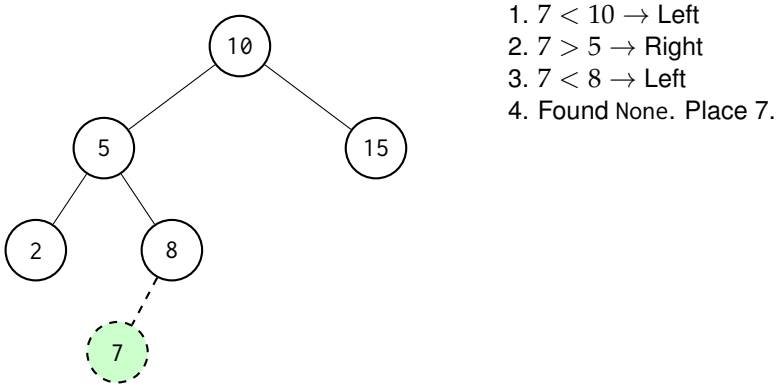


Figure 9.14: BST Insertion. The new value 7 "trickles down" to the correct empty spot.

9.9.4 Deletion in a BST ($O(h)$)

Deletion is the most complex operation. We must handle three cases:

- **Leaf Node:** Simply return `None` (remove it).
- **One Child:** Return its child (bypass the current node).
- **Two Children:** This is the tricky case. We cannot simply delete the node because it has two subtrees.
  1. Find its **in-order successor** (the smallest value in the Right Subtree).
  2. Replace the current node's value with the Successor's value.
  3. Recursively **delete** the Successor from the Right Subtree.

Note that there are various ways to handle deletion, this is simply one method. You will explore other methods in your missions.

```python
def get_min(tree):
    """Finds the smallest value (leftmost node)"""
    left, val, right = tree
    if left is None:
        return val
    return get_min(left)

def bst_delete(tree, target):
    if tree is None: return None
    left, val, right = tree
    if target < val:
        return (bst_delete(left, target), val, right)
    elif target > val:
        return (left, val, bst_delete(right, target))
    else:
        # Case 1 & 2: 0 or 1 child
        if left is None:
            return right
        if right is None:
            return left
        # Case 3: 2 children
        successor_val = get_min(right)
        new_right = bst_delete(right, successor_val)
        return (left, successor_val, new_right)
```

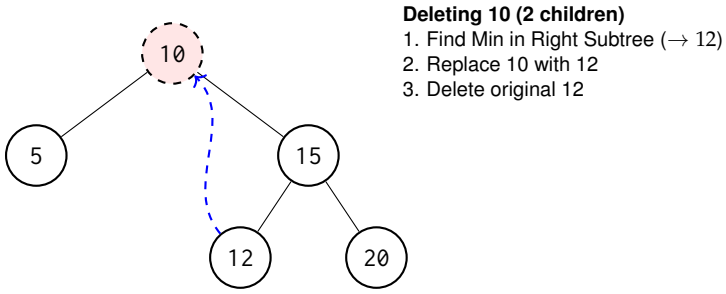


Figure 9.15: BST Deletion (Case 3). The successor (12) promotes itself to replace deleted roots.

9.9.5 Advanced: Self-Balancing Trees (Out of Syllabus)

A standard BST is vulnerable to the order of insertion. If you insert sorted data (e.g. 1, 2, 3, 4, 5), the tree degenerates into a **linked list** (skewed tree).

- **Skewed Tree Height:** $O(n) \implies$ Search/Insert/Delete become $O(n)$ (slow).
- **Balanced Tree Height:** $O(\log n) \implies$ Operations remain fast.

To guarantee $O(\log n)$ performance, we use **self-balancing BSTs** (e.g. AVL Trees, Red-Black Trees). These algorithms detect when the tree becomes "too deep" on one side and automatically perform **rotations** to fix the height.

Tree Rotations

A rotation is a local operation that changes the structure of the tree to reduce height while **preserving the BST Invariant**.

There are two fundamental types:

1. **Right Rotation:** Fixes a "Left-Heavy" tree (Left child moves up).
2. **Left Rotation:** Fixes a "Right-Heavy" tree (Right child moves up).

Logic of a Right Rotation (on node Y):

- Let X be the left child of Y.
- X becomes the new parent.
- Y becomes the *right* child of X.
- The right subtree of X (labeled B) is handed over to become the *left* subtree of Y.

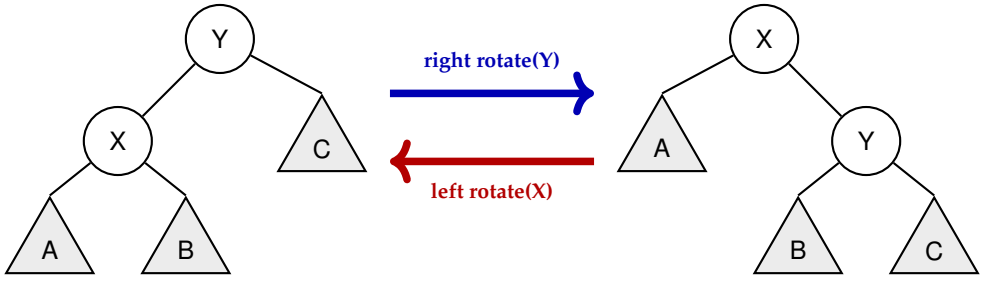


Figure 9.16: Tree Rotations. A Right Rotation moves the left child (X) up to become the parent, pushing the old parent (Y) down to the right. Subtree B transfers automatically.

AVL Rebalancing Logic

Named after its inventors **Adelson-Velsky and Landis**, an AVL tree guarantees performance by strictly enforcing a height constraint.

For every node, we calculate a **Balance Factor**:

$$\text{Balance} = \text{Height(Left Subtree)} - \text{Height(Right Subtree)}$$

In an AVL Tree, the balance factor of *every* node must be **-1, 0, or 1**.

- **0:** Perfectly balanced.
- **1 or -1:** Slightly unbalanced, but acceptable.
- > 1 **or** < -1**: Critically unbalanced**. The tree must immediately perform rotations to fix this.

When a node becomes critical (e.g. balance is 2 or -2), we check its child to decide which rotation to use:

- **Left-Left Case:** Parent is Left-Heavy (> 1) and Child is Left-Heavy. $\rightarrow$ **right rotation**.
- **Left-Right Case:** Parent is Left-Heavy (> 1) but Child is Right-Heavy. $\rightarrow$ **left rotate the child then right rotate the parent**.
- **Right-Right Case:** Parent is Right-Heavy (< -1) and Child is Right-Heavy. $\rightarrow$ **left rotation**.
- **Right-Left Case:** Parent is Right-Heavy (< -1) but Child is Left-Heavy. $\rightarrow$ **right rotate the child then left rotate the parent**.

We modify our tuple to cache the height: `(left, value, right, height)` to ensure these checks remain $O(1)$.

```
def get_height(node):
    if not node: return 0
    return node[3]

def get_balance(node):
    if not node: return 0
    return get_height(node[0]) - get_height(node[2])

def make_node(left, val, right):
    """Helper to create a node with updated height"""
    h = 1 + max(get_height(left), get_height(right))
    return (left, val, right, h)

def rotate_right(y):
    # Unpack Y (y_left is X)
    x, y_val, y_right, _ = y

    # Unpack X
    x_left, x_val, x_right, _ = x

    # Perform Rotation
    # New Y takes X's right child (T2) as its left child
```

```
    new_y = make_node(x_right, y_val, y_right)
    # New X takes New Y as its right child
    new_x = make_node(x_left, x_val, new_y)

    return new_x

def rotate_left(x):
    y, x_val, x_left, _ = x
    y_left, y_val, y_right, _ = y

    new_x = make_node(x_left, x_val, y_left)
    new_y = make_node(new_x, y_val, y_right)

    return new_y

def rebalance(node):
    balance = get_balance(node)
    left, val, right, h = node

    # Case 1: Left Heavy (Balance > 1)
    if balance > 1:
        # Check if Left Child is Right Heavy (Left-Right Case)
        if get_balance(left) < 0:
            left = rotate_left(left) # Convert to Left-Left
        return rotate_right((left, val, right, h))

    # Case 2: Right Heavy (Balance < -1)
    if balance < -1:
        # Check if Right Child is Left Heavy (Right-Left Case)
        if get_balance(right) > 0:
            right = rotate_right(right) # Convert to Right-Right
        return rotate_left(node)

    return node

def avl_insert(node, key):
    # 1. Standard BST Insert
    if not node:
        return make_node(None, key, None)
    left, val, right, h = node
    if key < val:
        left = avl_insert(left, key)
    elif key > val:
        right = avl_insert(right, key)
    else:
        return node # Duplicate
    # 2. Update Height & Rebalance
    # We create a temp node to recalc height, then rebalance it
    updated_node = make_node(left, val, right)
    return rebalance(updated_node)
```

9.10 General Tree Traversal

Unlike linear data structures (lists, linked lists) which have a single logical path, trees are non-linear. To "search" a tree, we must decide on an order to visit the nodes. The two fundamental strategies are **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**.

In this course, we often represent generic trees using nested tuples.

- **Internal Node:** A tuple acting as a container for other nodes, e.g. `((a, b), c)`.
- **Leaf Node:** An atomic value (integer, string) containing actual data, e.g. `'a'`.

Example: `tree = ((1, 2), (3, 4), 5)`

- Root has 3 children: A subtree `(1, 2)`, a subtree `(3, 4)`, and a leaf `5`.
- The leaves are the integers 1, 2, 3, 4, 5.

9.10.1 Depth-First Search (DFS)

DFS explores as **deeply** as possible along a branch before backtracking. This is sometimes known as a *pre-order* traversal.

The recursive structure of trees lends itself to a recursive implementation of DFS. We treat every node as the "root" of its own smaller subtree.

- **Base Case:** If the node is a leaf (an integer), return it wrapped in a list.
- **Recursive Step:** If the node is a tuple, loop through its children, call DFS on them, and combine the results.

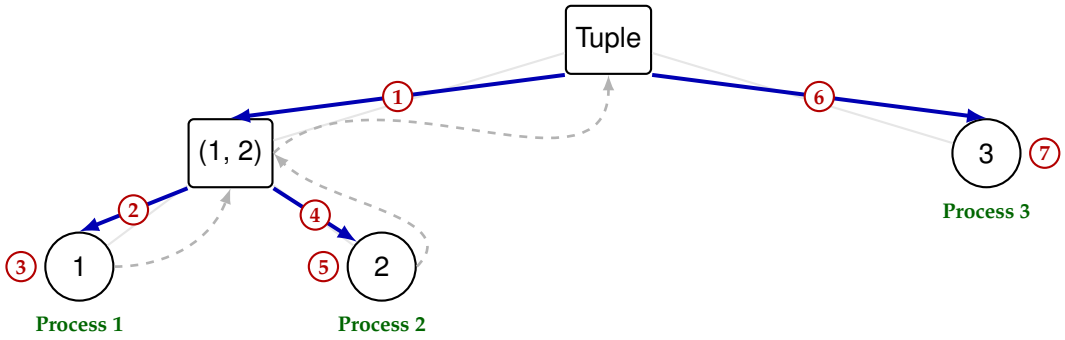
```
def dfs_recursive(tree):
    # 1. Base Case: Empty Tree
    if tree is None:
        return []
    # 2. Base Case: Leaf Node
    # If the item is not a tuple, it is a piece of data (e.g. 1, 2, 3)
    # We return it as a list so it can be combined with others.
    if not isinstance(tree, tuple):
        return [tree]
    # 3. Recursive Step: Internal Node (Tuple)
    result = []
    # Iterate through children left-to-right.
    # The recursion implicitly "pauses" the current function
    # and "dives" into the child function.
    for child in tree:
        child_result = dfs_recursive(child)
        result.extend(child_result)
    return result
```

We can also write it iteratively using a stack. To visit children in natural left-to-right order, we must push them onto the stack in **reverse** order (right child first, then left).

```python
def dfs_general(tree):
    if tree is None:
        return []
    stack = [tree]
    result = []
    while stack:
        curr = stack.pop() # LIFO: Pop the last added element
        if isinstance(curr, tuple):
            # Internal node, so extend the stack with children.
            # CRITICAL: Reverse order ensures left child popped first.
            for child in reversed(curr):
                stack.append(child)
        else:
            # Leaf node: Process it.
            result.append(curr)
    return result
```

Consider the `tree = ((1, 2), 3)`. DFS processes the nodes as follows:

Step	Action / Logic	Stack State	Output
1	Pop Root. It is a tuple. Push children **reversed**.	`[3, (1,2)]`	-
2	Pop `(1,2)`. It is a tuple. Push children **reversed**.	`[3, 2, 1]`	-
3 & 4	Pop 1. It is a leaf. **Process it**. Stack top is now 2.	`[3, 2]`	`[1]`
5 & 6	Pop 2. It is a leaf. **Process it**. Stack top is now 3.	`[3]`	`[1, 2]`
7	Pop 3. It is a leaf. **Process it**.	`[]`	`[1, 2, 3]`



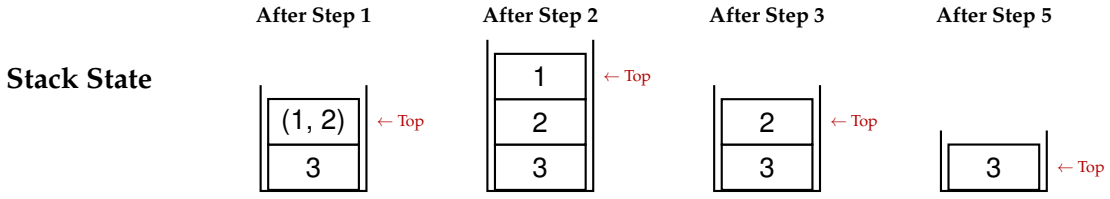


Figure 9.17: Visual Trace of DFS. Blue arrows indicate diving (pushing to stack). Dashed gray arrows indicate backtracking (popping).

9.10.2 Breadth-First Search (BFS)

BFS explores the tree **level-by-level**. It processes all immediate children before moving to grandchildren. Therefore, a queue (FIFO) is best suited for BFS.

```python
def bfs_general(tree):
    if tree is None:
        return []
    queue = [tree]
    result = []
    while queue:
        curr = queue.pop(0) # FIFO: Dequeue from the front
        if isinstance(curr, tuple):
            # Internal Node: Enqueue all children for future
                processing
            for child in curr:
                queue.append(child)
        else:
            # Leaf Node: Process immediately
            result.append(curr)

    return result
```

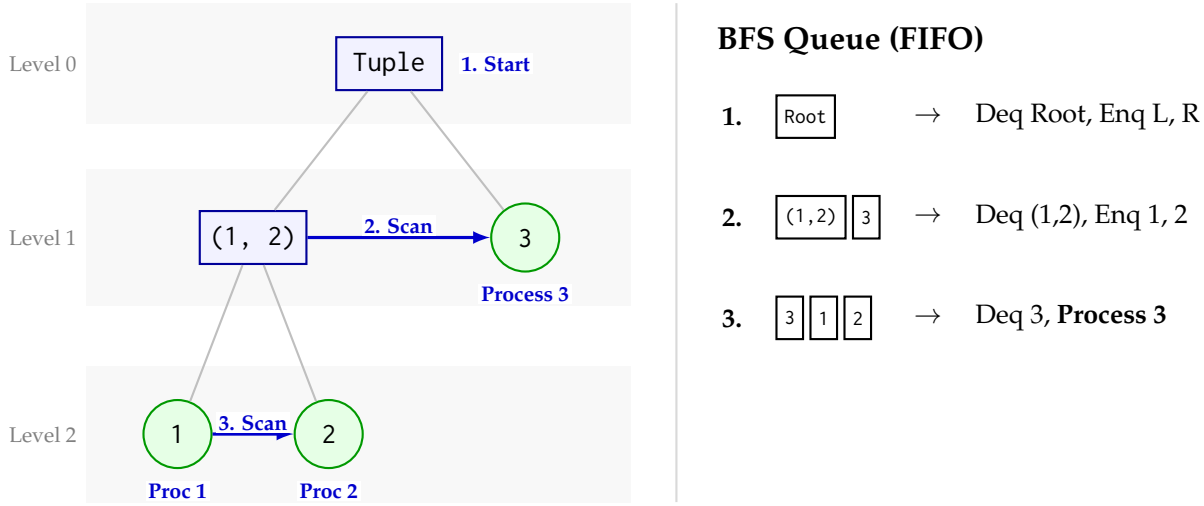


Figure 9.18: Visualizing BFS. The grey bands indicate depth levels. BFS scans horizontally across a level, processing leaves and enqueueing children of internal nodes, before moving deeper.

9.10.3 Grid Traversal and Flood Fill

A 2D Grid (Matrix) is a specialized graph structure frequently encountered in algorithmic problems. Unlike standard adjacency lists, the graph topology is implicit in the coordinate system:

- **Nodes:** Every cell (r, c) represents a node.
- **Edges:** Edges exist between a cell and its adjacent neighbors (typically Up, Down, Left, Right).

Key Distinction: Cycles

Unlike trees, grids contain cycles (e.g. $A \to B \to A$). To prevent infinite recursion, grid traversals **must** maintain a `visited` state. This is often done by using a dedicated `HashSet` or by modifying the grid in-place (e.g. changing `'1'` to `'0'` or `'V'`).

Just as we can do BFS and DFS on trees, we can use these to traverse grids too. We shall look at an application of it in a classic problem:

Given a binary grid where `'1'` represents land and `'0'` represents water, count the number of **islands**. An island is surrounded by water and formed by connecting adjacent lands horizontally or vertically.

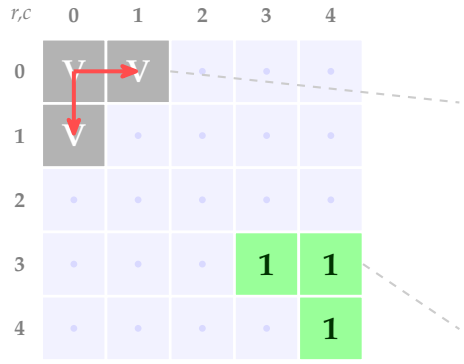
Algorithm Strategy (Flood Fill): We utilize a scan-and-traverse approach:

1. Iterate through every cell (r, c) in the grid.
2. If the cell is **land** (`'1'`) AND **unvisited**:
    - We have discovered a **new island**. Increment the counter.
    - Immediately trigger a traversal (DFS or BFS) to "sink" or mark the entire connected landmass. This ensures these cells are not recounted later.

Looking Forward: Sets

We will learn more about sets in future. For now, just understand that sets are used to efficiently check if a cell has already been visited.

```python
def count_islands(grid):
    if not grid: return 0
    rows, cols = len(grid), len(grid[0])
    visited = set()
    islands = 0
    def dfs_flood_fill(r, c):
        # 1. Bounds Check
        if r < 0 or r >= rows or c < 0 or c >= cols:
            return
        # 2. Validity Check (Water or Already Visited)
        if grid[r][c] == '0' or (r, c) in visited:
            return
        # 3. Mark Visited
        visited.add((r, c))
        # 4. Recurse (4-Directional)
        directions = [(1,0), (-1,0), (0,1), (0,-1)]
        for dr, dc in directions:
            dfs_flood_fill(r + dr, c + dc)
    # Main Scan
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '1' and (r, c) not in visited:
                islands += 1     # Found new island
                dfs_flood_fill(r, c) # Consume entire component
    return islands
```



Island 1 (Active DFS)

1. Main loop hits `(0,0)`.
2. `dfs(0,0)` triggered.
3. Recursive spread:
   - `(0,0)` → `(0,1)`
   - `(0,0)` → `(1,0)`
4. All marked **'V'** (Visited).

Island 2 (Pending)

- Currently holds `'1'`.
- Will be found when loop reaches `r=3, c=3`.
- `islands` count → 2.

Figure 9.19: Visualizing Grid DFS. The algorithm treats the grid as a graph where adjacent '1's are connected edges. The red arrows demonstrate the recursive spread of the flood fill operation.

9.11 Sorting

Sorting rearranges elements into an order (e.g. ascending) according to a comparison rule.

9.11.1 Stability and Memory

Various sorting algorithms have trade off in terms of two key ideas, stability and memory ("in-place").

1. What is a "Stable" Sort?

A sorting algorithm is considered **stable** if it preserves the relative order of elements that compare as equal. Stability is crucial when performing multiple passes of sorting on complex data.

Example: If you sort a list of students by *Name* (A-Z) and then sort them by *Grade*, a stable sort guarantees that students with the same Grade remain sorted by Name.

```python
# List of tuples: (Name, Grade)
students = [
    ("Alice", "B"),
    ("Bob",   "A"),
    ("Carol", "B"),
    ("Dave",  "A")
]

# 1. First Pass: Sort by Name (Already sorted, but for demonstration)
students.sort(key=lambda x: x[0])

# 2. Second Pass: Sort by Grade
# Since 'Alice' and 'Carol' both have 'B', a STABLE sort ensures
# 'Alice' stays before 'Carol' because that was their previous order.
students.sort(key=lambda x: x[1])

print(students)
# Output:
# [('Bob', 'A'), ('Dave', 'A'), ('Alice', 'B'), ('Carol', 'B')]
# Note: Bob comes before Dave, and Alice comes before Carol.
# The relative name order was preserved within the grade groups.
```

Note: Both `.sort()` *and* `sorted()` *are guaranteed to be stable in Python.*

2. What is an "In-Place" Sort?

An in-place algorithm updates the input data structure directly rather than creating a copy of the data.

- **`list.sort()`**: This is an **in-place** operation.
  - **Memory:** It is more memory efficient because it does not duplicate the list data.
  - **Risk:** It overwrites the original order. If you need the original data later, you must manually copy it first.
- **`sorted()`**: This is **NOT** in-place.
  - **Memory:** It allocates memory for a brand new list of size N.
  - **Safety:** The original variable remains untouched, making this safer for functional programming styles where side effects are discouraged.

```
# IN-PLACE (.sort)
# No new variable is created. The object at memory address 'id(a)' changes.
a = [3, 1, 2]
a.sort()
# a is now [1, 2, 3]

# NOT IN-PLACE (sorted)
# A new object is created at a different memory address.
b = [3, 1, 2]
c = sorted(b)
# b is still [3, 1, 2]
# c is [1, 2, 3]
```

9.11.2 Python Built-ins: `.sort()` vs. `sorted()`

Python offers two primary mechanisms for sorting. While both utilize the same efficient underlying algorithm (Timsort), they differ fundamentally in their operation regarding memory usage and return values.

1. `list.sort(key=None, reverse=False)`

This method sorts a list **in-place**. It modifies the original list directly and returns `None`.

- **Mutation:** The original order is overwritten. No new list object is created, making it more memory-efficient for large lists where preserving the original order is unnecessary.
- **Constraint:** This method is available *only* for list objects.
- **Common Pitfall:** Assigning the result of this method to a variable (e.g. `x = data.sort()`) will result in `x` being `None`.

2. `sorted(iterable, key=None, reverse=False)`

This built-in function creates and returns a **new list** containing the sorted elements from the provided iterable.

- **Non-Destructive:** The original data remains unchanged.
- **Universal:** Accepts *any* iterable, including lists, tuples, dictionaries, strings, sets, and generators.
- **Return Value:** Always returns a new list object.

Syntax & Parameters

Both methods support two powerful keyword-only arguments that allow for customized sorting logic.

`key` (callable, optional) A function that is called on each element prior to comparison. The sort is performed based on the *return value* of this function, not the element itself.

- **Default:** `None` (compares elements directly).
- **Usage:** Essential for complex objects or non-standard sorting (e.g. sorting strings by length, case-insensitive sorting).

`reverse` (bool, optional) A boolean flag to control the sort order.

- **Default:** `False` (Ascending order: 0-9, A-Z).
- **Usage:** Set to `True` for descending order.

Code Examples

```python
# --- 1. Basic Difference: In-Place vs New List ---
data = [4, 1, 3, 2]

# sorted() creates a copy
sorted_copy = sorted(data)
print(data)         # Output: [4, 1, 3, 2] (Unchanged)
print(sorted_copy)  # Output: [1, 2, 3, 4] (Sorted)

# .sort() modifies the original
data.sort()
print(data)         # Output: [1, 2, 3, 4] (Mutated)

# --- 2. Advanced Usage: Key and Reverse ---
words = ["banana", "Apple", "cherry", "date"]

# Sort by Length (shortest to longest)
# The 'key' function (len) is applied to each element.
# Comparisons are effectively: len("date") vs len("Apple")...
by_len = sorted(words, key=len)
# Result: ['date', 'Apple', 'banana', 'cherry']

# Case-Insensitive Sort
# Using str.lower ensures 'Apple' is treated as 'apple'
# Without this, 'Apple' comes before 'banana' due to ASCII values.
case_insensitive = sorted(words, key=str.lower)
# Result: ['Apple', 'banana', 'cherry', 'date']

# Descending Sort
nums = [10, 5, 20, 1]
nums.sort(reverse=True)
print(nums)
# Result: [20, 10, 5, 1]
```

Note

Python uses Timsort, a hybrid stable sort derived from merge sort and insertion sort.

$$\text{Time Complexity: } O(n \log n) \text{ (Worst/Average)}, \quad O(n) \text{ (Best - nearly sorted)}$$

Both methods are **stable**, meaning that if two elements have equal keys, their original relative order is preserved. This is crucial for multi-pass sorting (e.g. sort by date, then by name).

9.11.3 Sorting Records With Keys

We often want to sort complex data, and therefore, we will choose to sort them by keys. We can select what field to sort by making use of lambda functions. For example, in the example below, suppose we have records of the form `(name, grade, score)`, and we would like to sort them in descending order by their score. We can do so by using the `key` parameter with a lambda function to select the age from a given record.

```
students = [
    ('john',  'A', 15),
    ('jane',  'B', 10),
    ('ben',   'C',  8),
    ('simon', 'A', 21),
    ('dave',  'B', 12)
]

# sort by score descending (field index 2)
students.sort(key=lambda s: s[2], reverse=True)
```

9.11.4 Complexity Overview

There are some important sorting algorithms that we will cover. A summary of these sorts is given below:

Algorithm	Best	Average	Worst	Extra Space	Notes
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Best-case requires early-exit optimization.
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Few swaps; comparisons dominate.
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Excellent when nearly sorted; stable.
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Stable; typical implementation not in-place.
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place partition; worst-case from bad pivots.
Tree Sort (BST)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$	Worst-case if BST height becomes n.
Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	Needs small integer range $0..k$.

9.11.5 Bubble Sort

Concept: The algorithm repeatedly iterates through the list, comparing adjacent elements and swapping them if they are in the wrong order. This process causes the largest unsorted element to "bubble" to its correct position at the end of the list after each pass.

Complexity Analysis

- **Time Complexity:**
  - **Worst Case ($O(n^2)$):** Occurs when the array is reverse sorted. Every pair comparison requires a swap.
  - **Best Case ($O(n)$):** Occurs when the array is already sorted. By using a `swapped` flag, the algorithm can terminate early after one pass with no swaps.
- **Space Complexity:** $O(1)$ (In-place).
- **Stability:** Stable (equal elements are never swapped).

The implementation is as follows:

```
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        swapped = False
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # Optimization: If no two elements were swapped, break
        if not swapped:
            break
```

Stability

Note that if we were to change the comparison to >=, it would no longer be a stable sort.

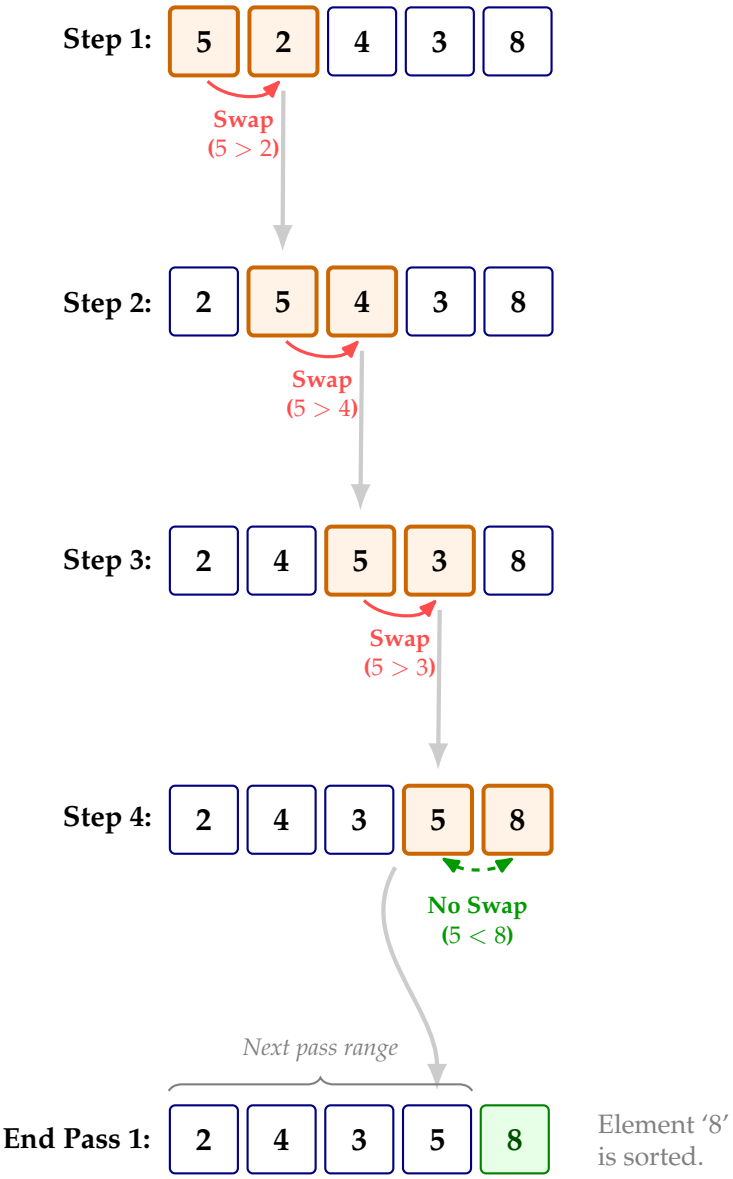


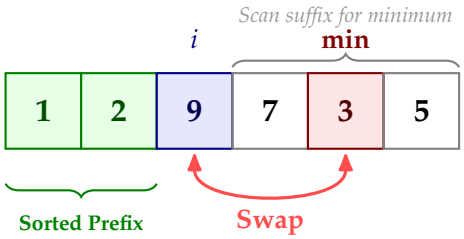
Figure 9.20: Visualizing one full pass of Bubble Sort.

9.11.6 Selection Sort

Concept: The algorithm divides the input list into two parts: a sorted sublist of items which is built up from left to right, and a remaining unsorted sublist. In each iteration, it finds the **minimum** element in the unsorted sublist and swaps it with the leftmost unsorted element.

Complexity Analysis

- **Time Complexity:** Always $O(n^2)$.
  - Even if the array is sorted, the algorithm must scan the entire unsorted suffix to confirm that the current element is indeed the minimum.
  - It performs $O(n^2)$ comparisons but only $O(n)$ swaps.
- **Space Complexity:** $O(1)$ (In-place).
- **Stability:** Not stable (long-distance swaps can reorder equal elements).



Current Step:
1. Current position is i (value 9).
2. Scanned suffix found **min** (value 3).
3. Performing **swap**.

Figure 9.21: Selection Sort: The algorithm identifies the sorted prefix (green), picks the current slot i (blue), scans the remaining unsorted suffix to find the minimum value (red), and swaps them.

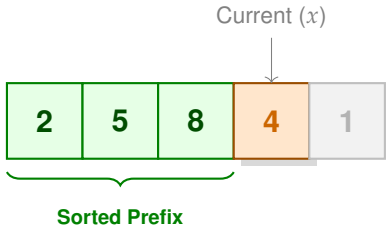
```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        # Find the minimum element in remaining unsorted array
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        # Swap the found minimum element with the first element
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

9.11.7 Insertion Sort

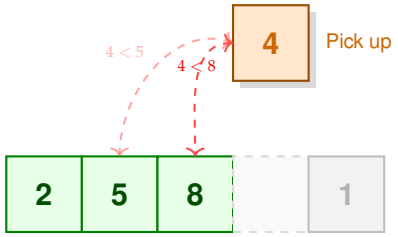
Concept: The algorithm builds the final sorted array one item at a time. It assumes the first element is sorted, then picks the next element and shifts elements in the sorted portion to the right to make space for the new item.

Complexity Analysis

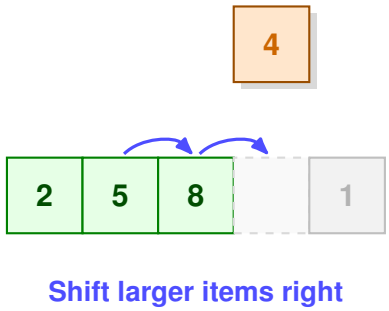
- **Time Complexity:**
  - **Best Case ($O(n)$):** Array is already sorted. The inner `while` loop never runs.
  - **Worst Case ($O(n^2)$):** Array is reverse sorted. Every insertion requires shifting all sorted elements.
- **Space Complexity:** $O(1)$ (In-place).



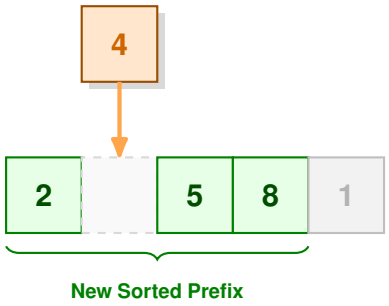
(a) **Step 1: Setup.** Identify sorted prefix and element $x = 4$.



(b) **Step 2: Compare.** Compare x backwards against sorted items.



(c) **Step 3: Shift.** Move elements $> x$ right.



(d) **Step 4: Insert.** Place x into the gap.

Figure 9.22: Visualizing Insertion Sort: Inserting value 4 into the sorted prefix $[2, 5, 8]$.

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        # Move elements of arr[0..i-1] that are greater than key
        # to one position ahead of their current position
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

9.11.8 Merge Sort

Concept: Merge Sort uses *Divide and Conquer* paradigm. It splits the problem into smaller subproblems (Divide), solves them recursively (Conquer), and combines the results.

Complexity Analysis

The time complexity is determined by the recurrence relation:

$$T(n) = 2T(n/2) + \Theta(n)$$

- $2T(n/2)$ represents the time to sort two subarrays of size $n/2$.
- $\Theta(n)$ represents the linear time required to **merge** the two sorted halves.

1. **Tree Height ($\log n$):** At every step, we divide the array size by 2. The number of times we can divide n by 2 until we reach 1 is $\log_2 n$.
2. **Work per Level (n):**
   - **Level 0:** 1 merge of size $n \rightarrow$ Cost: cn
   - **Level 1:** 2 merges of size $n/2 \rightarrow$ Cost: $2 \times (cn/2) = cn$
   - **Level 2:** 4 merges of size $n/4 \rightarrow$ Cost: $4 \times (cn/4) = cn$
   - **Level k:** 2^k merges of size $n/2^k \rightarrow$ Total cost is always cn.
3. **Total Cost:** Since there are $\log n$ levels and each level costs $O(n)$:

$$\text{Total Time} = (\text{Height}) \times (\text{Work per Level}) = (\log n) \times (n) = O(n \log n)$$

- **Space Complexity:** $O(n)$. Merge sort requires allocating new arrays during the merge phase ($O(n)$ auxiliary space). This makes it less space-efficient than In-Place Quicksort.

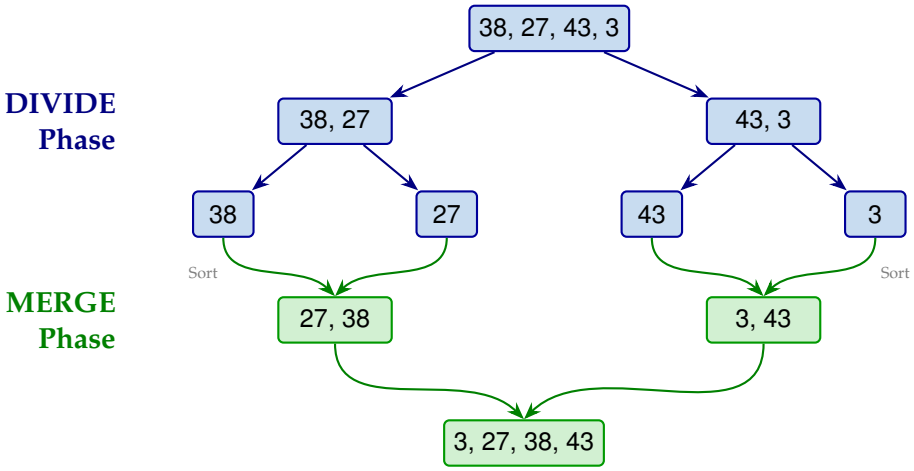


Figure 9.23: Visualizing Merge Sort: The array is recursively split into single elements (Blue), then merged back together in sorted order (Green).

It can be implemented as follows:

```python
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # Append any remaining items (one list will be empty)
    result.extend(left[i:])
    result.extend(right[j:])
    return result

def merge_sort(arr):
    if len(arr) <= 1: # Base Case: Single element is already sorted
        return arr
    mid = len(arr) // 2
    # Recursive Step: Divide O(log n)
    left_sorted = merge_sort(arr[:mid])
    right_sorted = merge_sort(arr[mid:])
    # Merge Step: Combine
    return merge(left_sorted, right_sorted)
```

The `merge` function is a crucial part of merge sort in order to combine two smaller solutions into a larger, correct solution. Crucially, the merge routine assumes that the two input lists (`left` and `right`) are **already sorted**. This allows us to merge them efficiently without rescanning the entire data.

Now, we can merge them using a two pointer method. As an analogy, imagine two stacks of sorted playing cards face up on a table. To create a single sorted stack, you only need to look at the top card of each stack.

1. **Initialize Pointers:** We place pointers at the start of both lists: i for `left` and j for `right`.
2. **Compare and Select:** We compare the values at `left[i]` and `right[j]`.
   - If `left`$[i] \leq$ `right`$[j]$, we append `left`$[i]$ to our `result` list and increment i.
   - Otherwise, we append `right`$[j]$ and increment j.
3. **Repeat:** We repeat step 2 until one of the pointers reaches the end of its respective list.
4. **Collect Remainder:** Once one list is exhausted, we know the remaining elements in the other list are already sorted and are all larger than what we have processed so far. We simply append (extend) the rest of that non-empty list to `result`.

Because we pass through the lists exactly once, performing a constant number of comparisons per element, the time complexity of a single merge operation is linear:

$$O(n) \quad \text{where } n = \text{len(left)} + \text{len(right)}$$

This linear efficiency in the combine step is critical. If the merge step were slower (e.g., $O(n^2)$), the overall recursive algorithm would fail to achieve the optimal $O(n \log n)$ performance.

9.11.9 Quicksort

Concept: A divide-and-conquer algorithm that selects a **pivot** element x and partitions the array such that elements $< x$ pivot are on the left and elements $> x$ are on the right.

Complexity Analysis

- **Time Complexity:**
  - **Average Case ($O(n \log n)$):** Occurs when the pivot roughly bisects the array.
  - **Worst Case ($O(n^2)$):** Occurs if the pivot is always the smallest or largest element (e.g., sorting an already sorted array with the first element as pivot). This can be mitigated by choosing a random pivot or the "median-of-three".
- **Space Complexity:** $O(\log n)$ (Stack space for recursion).

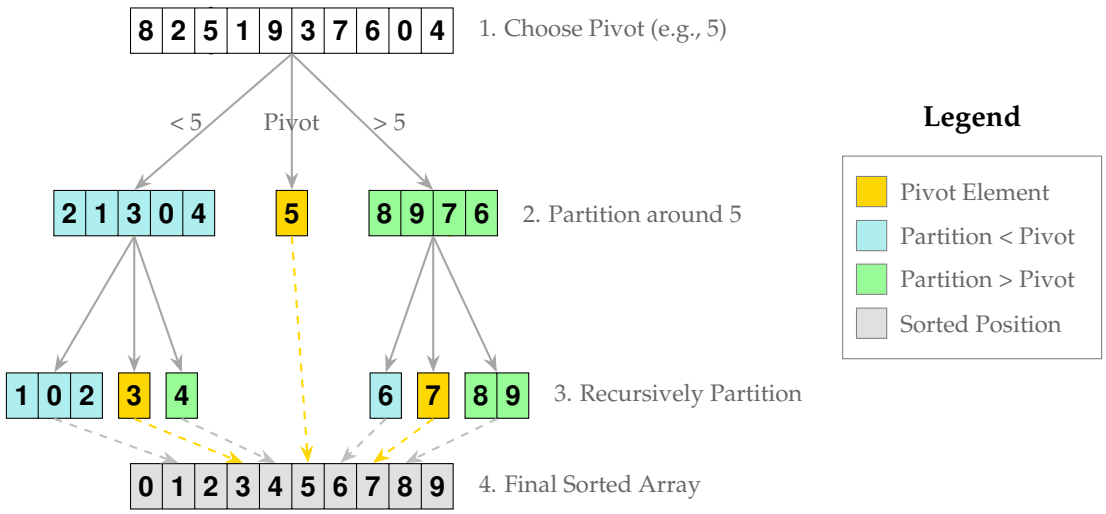


Figure 9.24: Quicksort Partitioning: Elements are reordered around the pivot.

```python
def partition(arr, low, high):
    pivot = arr[high]  # Choosing last element as pivot
    i = low - 1        # Pointer for greater element
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)
```

9.11.10 Tree Sort

Concept: Tree Sort is a comparison-based sorting algorithm that utilizes the properties of a **Binary Search Tree (BST)**. The algorithm operates in two phases:

1. **Construction:** All elements from the unsorted input are inserted into a BST. By definition, for any node tuple (L, V, R), all values in the left subtree L are strictly less than V, and all values in the right subtree R are strictly greater than V.
2. **Traversal:** An *in-order traversal* (which can be implemented with a variant of DFS) is performed on the constructed tree. This traversal visits nodes in the order `Left` $\rightarrow$ `Root` $\rightarrow$ `Right`, which inherently retrieves the elements in sorted ascending order.

Complexity Analysis

The efficiency of Tree Sort depends heavily on the shape (height h) of the resulting BST.

- **Time Complexity:**
  - **Average Case ($O(n \log n)$):** If elements are inserted in random order, the tree tends to be balanced, with height $h \approx \log n$.
  - **Worst Case ($O(n^2)$):** If the input is already sorted (or reverse-sorted), the tree degenerates into a linear chain (linked list) where $h = n$.
- **Space Complexity:** $O(n)$
  - Requires allocation of n nodes (or tuples) to store the data structure.

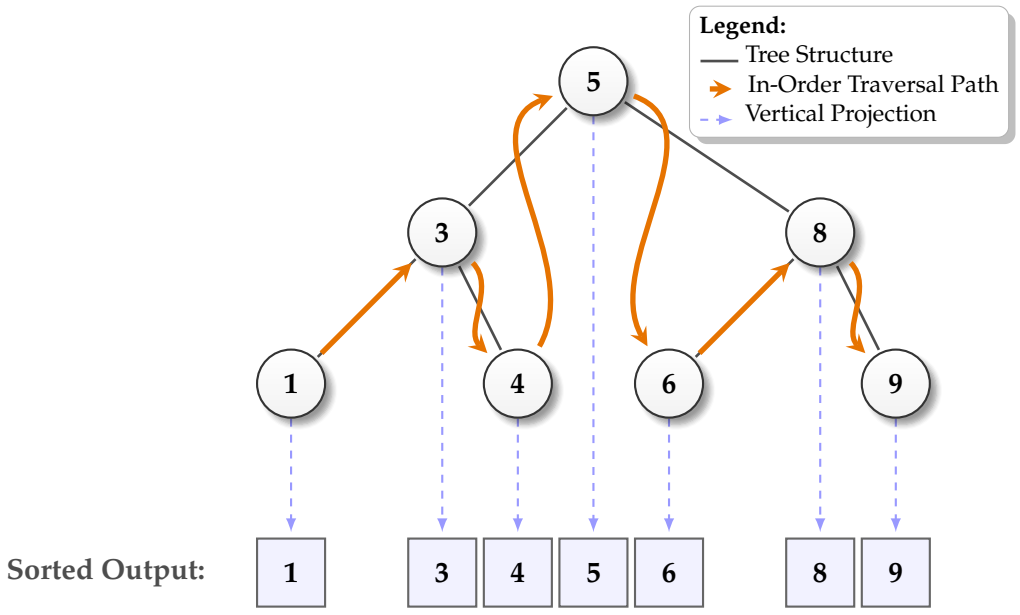


Figure 9.25: Visualizing Tree Sort: The structure of a BST allows elements to be "projected" vertically onto a sorted number line.

This implementation uses **tuples** to represent tree nodes in the format `(left_subtree, value, right_subtree)`, as with the earlier implementation of BSTs.

```python
def insert(tree, key):
    """
    Inserts a key into the BST (represented as tuples).
    Tuple structure: (left, value, right)
    """
    # Base Case: Create a new leaf tuple
    if tree is None:
        return (None, key, None)

    left, val, right = tree

    # Recursively insert into the correct subtree
    if key < val:
        return (insert(left, key), val, right)
    else:
        return (left, val, insert(right, key))

def inorder_traversal(tree):
    """
    Traverses the tree: Left -> Root -> Right
    """
    if tree is None:
        return []

    left, val, right = tree
    # Concatenate: Left List + [Current] + Right List
    return inorder_traversal(left) + [val] + inorder_traversal(right)

def tree_sort(arr):
    tree = None
    # Phase 1: Construct the BST
    for key in arr:
        tree = insert(tree, key)

    # Phase 2: Flatten tree to get sorted elements
    return inorder_traversal(tree)
```

Notice that we have made use of quite a few algorithms that we have previously written to write this sort. This is the power of abstractions, where we can combine smaller pieces of code to build larger ones.

9.11.11 Counting Sort

Concept: Counting sort is a non-comparison-based sorting algorithm. It operates by counting the number of objects that possess distinct key values. This technique essentially maps input values to indices in an auxiliary frequency array, allowing for sorting in linear time relative to the number of items and the range of input values.

Complexity Analysis

- **Time Complexity:** $O(n + k)$
  - n is the number of elements in the input array.
  - k is the range of the input (difference between max and min values).
  - The algorithm iterates over the input (n) and the count array (k).
- **Space Complexity:** $O(n + k)$
  - Requires auxiliary space for the count array (k) and the output array (n).
- **Constraints:**
  - Efficient only when k is not significantly larger than n (e.g., $k \approx n$).
  - Generally restricted to integers or data that can be mapped to integer keys.

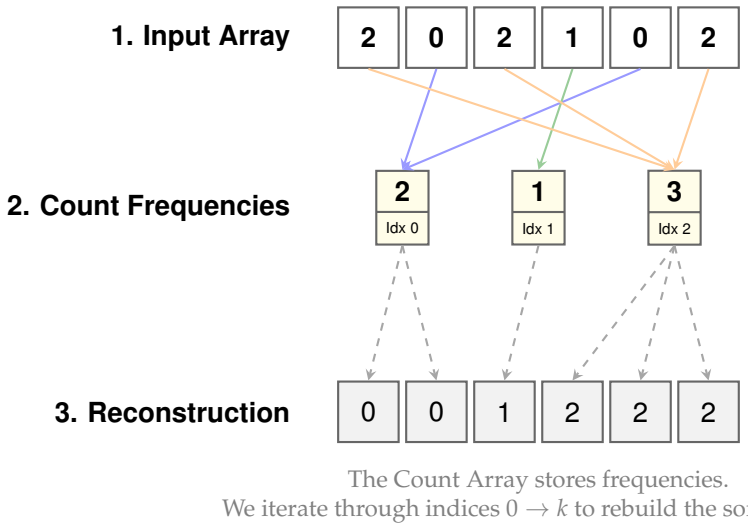


Figure 9.26: Visualizing Counting Sort. Input values map to indices in the Count Array, which are then expanded to form the sorted output.

The following implementation demonstrates the basic frequency-counting approach.

```python
def counting_sort(arr):
    if not arr:
        return []

    # 1. Find the range of input data
    max_val = max(arr)

    # 2. Initialize count array with zeros
    # Size is max_val + 1 to accommodate 0-based indexing
    count = [0] * (max_val + 1)

    # 3. Populate the count array
    for num in arr:
        count[num] += 1

    # 4. Rebuild the sorted array
    sorted_arr = []
    for i, freq in enumerate(count):
        # Append the index 'i' repeatedly based on its frequency
        sorted_arr.extend([i] * freq)

    return sorted_arr

# Example Usage
data = [4, 2, 2, 8, 3, 3, 1]
print(counting_sort(data))
# Output: [1, 2, 2, 3, 3, 4, 8]
```