DATA STRUCTURES & ALGORITHMS

A practical guide to organizing information and designing efficient solutions

LEARNSAPPEAL | ORIGINAL COURSE

Choose the right structure. Make each step count.

Explore arrays, maps, stacks, queues, trees, graphs, searching, sorting, and problem-solving patterns with Python examples.

What you need

Basic Python helps: variables, loops, functions, and lists. Each idea is explained before its code example. Trace small inputs by hand, then run the example and test a few edge cases.

Course map: Data structures (Modules 1-6), then algorithms and problem-solving patterns (Modules 7-12). Includes code, complexity intuition, questions with answers, projects, and a glossary.

How to use this course

For each problem, write down what the input and output should be. Pick a structure that makes the operations you need convenient. First write a clear solution; then reason about its running time and memory. Test empty inputs, one item, duplicates, and already sorted or reversed data.

By the end, you can: compare common structures; read Big-O as growth rather than exact seconds; search and sort data; traverse trees and graphs; use heaps; recognize greedy and dynamic programming patterns; and explain trade-offs in a solution.

Course map

• 1. Problem solving and complexity
• 2. Arrays, strings, and hash maps
• 3. Stacks and queues
• 4. Linked lists
• 5. Recursion and divide and conquer
• 6. Searching and sorting
• 7. Trees and binary search trees
• 8. Heaps and priority queues
• 9. Graphs: BFS and DFS
• 10. Shortest paths and topological order
• 11. Greedy and dynamic programming
• 12. Capstone: route and task planner

DATA STRUCTURES

1. Problem solving and complexity

An algorithm is a precise sequence of steps. Before coding, clarify the input, expected output, constraints, and edge cases. Big-O describes how resource use grows as input size n grows. O(1) is constant, O(log n) grows slowly, O(n) scans items, and O(n squared) often compares every pair. It is a growth model, not a stopwatch.

Worked example

```
def contains_value(items, target):
    for item in items:
        if item == target:
            return True
    return False
# One scan: O(n) time, O(1) extra space
```

Check your understanding

- What does Big-O describe?
- What is the time complexity of one full list scan?

Suggested answers

1. How time or memory use grows with input size.
2. O(n).

2. Arrays, strings, and hash maps

Python lists are dynamic arrays: indexing is fast, while inserting near the front shifts later values. Strings are immutable sequences of characters. Dictionaries use hashing to map keys to values and usually provide fast lookup. Sets track unique values. Choose based on required operations, not only how data looks.

Worked example

```
names = ['Asha', 'Bo', 'Chen']
first = names[0]          # indexed access
counts = {'web': 4, 'python': 7}
seen = set(names)
print(counts.get('web', 0), 'Bo' in seen)
```

Check your understanding

- Which structure maps a key to a value?
- Why can insertion at the front of a list be costly?

Suggested answers

1. A dictionary (hash map).
2. Existing items may need to shift.

3. Stacks and queues

A stack is last-in, first-out (LIFO), useful for undo history and matching brackets. A queue is first-in, first-out (FIFO), useful for work waiting its turn. Python lists work well as stacks with append and pop. For queues, collections.deque provides efficient operations at both ends.

Worked example

```
from collections import deque
stack = []
stack.append('page A')
stack.append('page B')
last = stack.pop()
queue = deque(['job A', 'job B'])
next_job = queue.popleft()
```

Check your understanding

- Which item leaves a stack first?
- What order does a queue follow?

Suggested answers

1. The most recently added item.
2. First-in, first-out.

4. Linked lists

A linked list stores values in nodes, where each node points to the next. Inserting after a known node can be constant time, but reaching the middle requires walking node by node. Python lists are usually a better default for general work; linked lists are useful to understand pointer-based structures and some specialized designs.

Worked example

```
class Node:
    def __init__(self, value, next_node=None):
        self.value = value
        self.next = next_node

head = Node('first', Node('second'))
current = head
while current is not None:
    print(current.value)
    current = current.next
```

Check your understanding

- What does a node store besides its value?
- Why is indexing into a linked list slow?

Suggested answers

1. A link to another node.
2. It must follow links from the start one node at a time.

5. Recursion and divide and conquer

A recursive function solves a problem by calling itself on a smaller input. It needs a base case to stop. Divide and conquer splits a problem, solves smaller parts, then combines results. Recursion can make tree logic clear, but deep recursion may exceed the language call limit; an explicit stack can avoid that.

Worked example

```
def factorial(n):
    if n <= 1:             # base case
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # 120
```

Check your understanding

- What is a base case?
- What happens if recursive inputs never get smaller?

Suggested answers

1. A condition that returns without another recursive call.
2. The calls may continue until the program fails.

6. Searching and sorting

Linear search checks items one at a time and works on unsorted data. Binary search repeatedly halves a sorted search space, taking O(log n) time. Sorting costs time but can make later searches or reports easier. Python sorted() returns a new list; list.sort() changes a list in place. A key function customizes ordering.

Worked example

```
from bisect import bisect_left
scores = [12, 18, 25, 31, 44]  # sorted
index = bisect_left(scores, 25)
found = index < len(scores) and scores[index] == 25
names = ['Mina', 'Ari', 'Zed']
print(sorted(names, key=str.lower))
```

Check your understanding

- What must be true before binary search?
- What is the difference between sorted() and list.sort()?

Suggested answers

1. The searched sequence must be sorted.
2. sorted returns a new list; sort modifies the list in place.

ALGORITHMS AND PROBLEM-SOLVING PATTERNS

7. Trees and binary search trees

A tree models hierarchy. It has a root, parent-child links, and leaves with no children. A binary tree has at most two children per node. In a binary search tree, smaller keys go left and larger keys go right; search can be efficient when balanced, but a badly skewed tree can behave like a list. Traversals visit nodes in systematic orders.

Worked example

```
class TreeNode:
    def __init__(self, key, left=None, right=None):
        self.key, self.left, self.right = key, left, right

def inorder(node):
    if node is None:
        return
    inorder(node.left)
    print(node.key)
    inorder(node.right)
```

Check your understanding

- What is a leaf?
- Why can an unbalanced search tree be slow?

Suggested answers

1. A node with no children.
2. It may become a chain, requiring a linear walk.

8. Heaps and priority queues

A heap is a tree-shaped ordering that keeps the smallest or largest item easy to retrieve. A min-heap supports inspecting the minimum in O(1), and pushing or popping in O(log n). It is not fully sorted. Priority queues use heaps when the next item should be chosen by priority.

Worked example

```
import heapq
jobs = [(3, 'email'), (1, 'backup'), (2, 'report')]
heapq.heapify(jobs)
priority, job = heapq.heappop(jobs)
print(job)  # backup
heapq.heappush(jobs, (0, 'urgent'))
```

Check your understanding

- Is a heap fully sorted?
- What is a priority queue useful for?

Suggested answers

1. No; it guarantees the top item has highest priority by its ordering.
2. Repeatedly selecting the next most important or smallest item.

9. Graphs: BFS and DFS

A graph has vertices connected by edges. It can model roads, links, dependencies, or friendships. Breadth-first search (BFS) explores in layers and finds a shortest path by number of edges in an unweighted graph. Depth-first search (DFS) follows a path before backtracking. A visited set prevents revisiting nodes forever.

Worked example

```
from collections import deque
graph = {'A': ['B', 'C'], 'B': ['D'], 'C': [], 'D': []}
def bfs(start):
    seen, q = {start}, deque([start])
    while q:
        node = q.popleft()
        print(node)
        for nxt in graph[node]:
            if nxt not in seen:
                seen.add(nxt); q.append(nxt)
```

Check your understanding

- Which search explores in layers?
- Why keep a visited set?

Suggested answers

1. BFS.
2. To avoid processing a node repeatedly or looping forever.

10. Shortest paths and topological order

For weighted graphs with nonnegative edge costs, Dijkstra's algorithm repeatedly expands the cheapest known route, commonly using a heap. Topological ordering arranges a directed acyclic graph so prerequisites come before dependents. These algorithms depend on assumptions: negative weights need other methods, and a cycle means no valid topological order exists.

Worked example

```
import heapq
# Start with distance zero; relax outgoing edges.
# Heap entries are (known_distance, node).
frontier = [(0, 'start')]
distance = {'start': 0}
# For each edge u -> v with cost w:
# candidate = distance[u] + w
# if candidate improves distance[v], update and push.
```

Check your understanding

- What restriction does Dijkstra commonly require?
- When does topological ordering apply?

Suggested answers

1. Nonnegative edge weights.
2. To directed acyclic graphs (DAGs).

11. Greedy and dynamic programming

A greedy algorithm makes the best-looking local choice at each step. It is correct only when the problem has structure that makes those choices safe. Dynamic programming stores answers to overlapping subproblems so they are not recomputed. Define the state, recurrence, base cases, and computation order before writing code.

Worked example

```
def ways_to_climb(n):
    if n <= 2:
        return n
    prev2, prev1 = 1, 2
    for _ in range(3, n + 1):
        prev2, prev1 = prev1, prev1 + prev2
    return prev1
print(ways_to_climb(5))  # 8
```

Check your understanding

- Why can a greedy choice fail?
- What does dynamic programming reuse?

Suggested answers

1. A locally best step may block the best overall solution.
2. Solutions to repeated subproblems.

12. Capstone: route and task planner

Build a small planner for a learning platform. Represent lessons as a dependency graph; find which lessons are reachable from a starting topic; use a topological order to suggest a valid sequence. Add priorities to a queue of study tasks. Explain the time and memory cost of each operation and test empty, disconnected, and cyclic inputs.

Worked example

```
Project plan:
1. Store lessons and prerequisite edges.
2. Use BFS to find reachable lessons.
3. Use topological sorting for a study sequence.
4. Use a heap for prioritized review tasks.
5. Detect cycles and explain invalid plans.
```

Check your understanding

- What does a cycle mean for prerequisite order?

• Which structure helps choose the next priority task?

Suggested answers

1. No valid topological order exists until the cycle is resolved.
2. A heap-backed priority queue.

Practice projects

1. Word frequency analyzer

Count words in a text file with a dictionary, ignore case, and report the most common terms. Compare one-pass counting with repeated scanning.

2. Support ticket scheduler

Represent tickets with priorities and use a heap to select the next ticket. Define a tie-breaking rule for equal priorities and test empty queues.

3. Campus route finder

Model locations as a graph. Use BFS when every route step has equal cost; use a weighted shortest-path method when travel costs differ. Explain the assumptions.

Project checklist
Are inputs and outputs clear? Is the chosen structure a good fit? Have you considered empty and boundary cases? Can you justify time and memory costs? Are assumptions stated?

Complexity quick reference

Operation / pattern	Typical growth	Notes
Index a dynamic array	O(1)	Access by position
Scan an array	O(n)	May inspect every item
Hash map lookup	Average O(1)	Worst case can be slower
Binary search	O(log n)	Input must be sorted
Comparison sort	O(n log n)	Typical efficient sorting
BFS / DFS	O(V + E)	Adjacency-list graph
Heap push / pop	O(log n)	Peek at root is O(1)

Glossary

Term	Meaning
Adjacency list	A graph representation listing neighbors for each vertex.
Algorithm	A precise sequence of steps for solving a problem.
Big-O	A description of how resource use grows with input size.
Deque	A double-ended queue with efficient operations at both ends.
Edge	A connection between graph vertices.
Hash map	A key-to-value structure, called dict in Python.
Heap	A structure that keeps an extreme value available at the root.
Node	An item in a linked structure or tree.
Recursion	A function solving a problem by calling itself on smaller inputs.
Stable sort	A sort that preserves the order of equal-key items.
Vertex	A node in a graph.
Dynamic programming	Solving overlapping subproblems by storing their answers.

Further study

Use the Python documentation to confirm library behavior and complexity notes for your implementation. These references cover common built-in structures and algorithms.

- Python Software Foundation. *Data structures.* https://docs.python.org/3/tutorial/datastructures.html
- Python Software Foundation. *collections — Container datatypes.* https://docs.python.org/3/library/collections.html
- Python Software Foundation. *heapq — Heap queue algorithm.* https://docs.python.org/3/library/heapq.html
- Python Software Foundation. *bisect — Array bisection algorithm.* https://docs.python.org/3/library/bisect.html
- Python Software Foundation. *Sorting HOW TO.* https://docs.python.org/3/howto/sorting.html

Course completion checklist

I can choose common data structures based on operations; estimate time and memory growth; use searching, sorting, trees, heaps, and graphs; recognize recursion, greedy, and dynamic programming patterns; and explain correctness assumptions and trade-offs.