Python Algorithm Library Reference

Comprehensive Guide to Built-in Algorithm Implementations
Generated: November 18, 2025

Introduction

This reference maps common interview algorithms and data structures to Python's standard library and popular third-party packages. Instead of implementing from scratch, leverage these battle-tested, optimized implementations.

Part 1: Python Standard Library

1. collections - High-Performance Data Structures

Data Structure	Use Case	Key Methods
Counter	Frequency counting, hash map	most_common(), elements()
defaultdict	Hash map with default values	Auto-initialization
deque	Queue, stack (O(1) both ends)	append(), appendleft(), pop(), popleft()
OrderedDict	Maintain insertion order (3.7+ dict does this)	move_to_end(), popitem()
ChainMap	Multiple dicts as single unit	maps, new_child()

Example: Counter for frequency counting

```
from collections import Counter
nums = [1, 1, 1, 2, 2, 3]
freq = Counter(nums)
top_k = freq.most_common(2) # [(1, 3), (2, 2)]
```

2. heapq - Heap/Priority Queue

Function	Purpose	Time Complexity
heappush(heap, item)	Add item to heap	O(log n)
heappop(heap)	Remove and return smallest	O(log n)
heapify(list)	Convert list to heap in-place	O(n)
nlargest(n, iterable)	Return n largest elements	O(n log k)
nsmallest(n, iterable)	Return n smallest elements	O(n log k)
heapreplace(heap, item)	Pop and push in one operation	O(log n)

Note: Python's heapq is a min-heap. For max-heap, negate values.

```
import heapq
# Min heap
min_heap = [3, 1, 4, 1, 5]
heapq.heapify(min_heap)

# Max heap (negate values)
max_heap = [-x for x in [3, 1, 4, 1, 5]]
heapq.heapify(max_heap)
largest = -heapq.heappop(max_heap)
```

3. bisect - Binary Search

Function	Purpose	Use Case
bisect_left(a, x)	Leftmost insertion point	Find first occurrence
bisect_right(a, x)	Rightmost insertion point	Find last occurrence + 1
insort_left(a, x)	Insert maintaining order (left)	Sorted insertion
insort_right(a, x)	Insert maintaining order (right)	Sorted insertion

```
import bisect
sorted_list = [1, 3, 3, 3, 5, 7]
# Find first occurrence of 3
left = bisect.bisect_left(sorted_list, 3) # 1
# Find position after last 3
right = bisect.bisect_right(sorted_list, 3) # 4
```

4. itertools - Combinatorics & Iteration

Function	Purpose	Example
combinations(it, r)	All r-length combinations	C(n,r) - no repeats
combinations_with_replacement	Combinations with repeats	Choose with replacement
permutations(it, r)	All r-length permutations	P(n,r) - order matters
product(*iterables)	Cartesian product	Nested loops
accumulate(it, func)	Running totals	Prefix sums
chain(*iterables)	Flatten iterables	Concatenate sequences
groupby(it, key)	Group consecutive elements	Group by key function

5. functools - Function Tools

Function	Purpose	Interview Use
lru_cache(maxsize)	Memoization decorator	DP problems, cache results
cache	Unlimited memoization (3.9+)	Simple DP caching
reduce(func, iterable)	Reduce to single value	Accumulation operations
cmp_to_key(func)	Convert comparison to key	Custom sorting

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n <= 1: return n
    return fibonacci(n-1) + fibonacci(n-2)
```

6. math - Mathematical Functions

Function	Purpose	Complexity
gcd(a, b)	Greatest common divisor	O(log min(a,b))
lcm(a, b)	Least common multiple (3.9+)	O(log min(a,b))
factorial(n)	n! factorial	O(n)
comb(n, k)	Binomial coefficient C(n,k)	O(k)
perm(n, k)	Permutations P(n,k)	O(k)
isqrt(n)	Integer square root	O(1)
ceil(x), floor(x)	Ceiling and floor	O(1)

Part 2: Algorithm Pattern to Library Mapping

Arrays & Hash Map Patterns

Pattern/Problem	Python Library Solution	Manual Implementation
Frequency counting	collections.Counter	dict with get()
Two Sum (sorted)	bisect module	Two pointers
Top K elements	heapq.nlargest()	Min heap of size k
Group anagrams	collections.defaultdict	dict with sorted key
Running sum/prefix	itertools.accumulate()	Manual accumulation
Median of stream	Use two heapq heaps	Max heap + Min heap

Binary Search Patterns

Pattern	Library Function	Notes
Find insertion point	bisect.bisect_left()	O(log n)
Find range	bisect_left() + bisect_right()	First and last occurrence
Insert in sorted order	bisect.insort()	O(n) due to insertion
Check if exists	index = bisect_left(); check a[index]	Then verify

Dynamic Programming Patterns

Pattern	Library Tool	Example
Memoization	@functools.lru_cache	Fibonacci, DFS with cache
Combinations count	math.comb(n, k)	Choose k from n
Permutations count	math.perm(n, k)	Arrange k from n
All combinations	itertools.combinations()	Generate all subsets
All permutations	itertools.permutations()	Generate all arrangements

Stack & Queue Patterns

Data Structure	Library Implementation	Operations
Stack	list with append()/pop()	O(1) both ends
Queue	collections.deque	O(1) both ends
Priority Queue (min)	heapq module	O(log n) push/pop
Priority Queue (max)	heapq with negated values	O(log n) push/pop
Deque (double-ended)	collections.deque	appendleft(), popleft()

Part 3: Third-Party Libraries

1. NumPy - Numerical Computing

Install: **pip install numpy**

Feature	NumPy Function	Use Case
Array operations	np.array(), np.zeros(), np.ones()	Efficient arrays
Matrix operations	np.dot(), np.matmul(), @	Linear algebra
Sorting	np.sort(), np.argsort()	O(n log n) optimized
Searching	np.searchsorted()	Binary search on arrays
Statistics	np.mean(), np.median(), np.std()	Quick stats
Unique elements	np.unique()	Deduplicate with counts
Argmax/Argmin	np.argmax(), np.argmin()	Find index of max/min

2. NetworkX - Graph Algorithms

Install: **pip install networkx**

Algorithm	NetworkX Function	Purpose
Shortest path	nx.shortest_path()	BFS/Dijkstra
All shortest paths	nx.all_shortest_paths()	Find all paths
Cycle detection	nx.find_cycle()	Detect cycles
Topological sort	nx.topological_sort()	DAG ordering
Connected components	nx.connected_components()	Find components
Minimum spanning tree	nx.minimum_spanning_tree()	Kruskal's/Prim's
Graph coloring	nx.greedy_color()	Vertex coloring
Centrality	nx.betweenness_centrality()	Important nodes

3. SciPy - Scientific Computing

Install: **pip install scipy**

Module	Key Functions	Algorithm
scipy.sparse	Sparse matrices	Memory-efficient matrices
scipy.spatial	KDTree, distance metrics	Nearest neighbors
scipy.optimize	minimize(), linear_sum_assignment()	Optimization
scipy.sparse.csgraph	shortest_path(), dijkstra()	Graph algorithms
scipy.special	comb(), perm()	Combinatorics

Part 4: Quick Problem-to-Library Mapping

Interview Problem	Python Library Solution
Two Sum (sorted array)	Use bisect.bisect_left() for each element
Top K Frequent Elements	collections.Counter + heapq.nlargest()
Group Anagrams	collections.defaultdict with sorted string key
Valid Anagram	collections.Counter comparison
Merge K Sorted Lists	heapq.merge(*lists)
Find Median from Stream	Two heaps: heapq for min, negate for max
Course Schedule	Manual (or use networkx.topological_sort)
Number of Islands	Manual BFS/DFS (or networkx for graphs)
Word Ladder	collections.deque for BFS
LRU Cache	@functools.lru_cache or manual OrderedDict
Binary Search	bisect.bisect_left() / bisect_right()
Permutations	itertools.permutations()
Combinations	itertools.combinations()
Subsets (Power Set)	itertools.combinations for each size
Coin Change (DP)	@functools.lru_cache for memoization
Climbing Stairs	@functools.lru_cache for memoization
GCD / LCM	math.gcd() / math.lcm()
Factorial	math.factorial()
Prefix Sum	itertools.accumulate()

Part 5: Complete Code Examples

Example 1: Top K Frequent Elements

```
from collections import Counter
import heapq

def top_k_frequent(nums, k):
    # Count frequencies
    freq = Counter(nums)
    # Get k most common
    return [num for num, count in freq.most_common(k)]

# Alternative with heapq
def top_k_heap(nums, k):
    freq = Counter(nums)
    return heapq.nlargest(k, freq.keys(), key=freq.get)
```

Example 2: Binary Search with bisect

```
import bisect

def search_range(nums, target):
    """Find first and last position of target"""
    left = bisect.bisect_left(nums, target)
    right = bisect.bisect_right(nums, target) - 1

    if left <= right:
        return [left, right]
    return [-1, -1]
```

Example 3: Memoization with functools

```
from functools import lru_cache

@lru_cache(maxsize=None)
def coin_change(amount, coins):
    """Min coins to make amount"""
    if amount == 0: return 0
    if amount < 0: return float('inf')

    min_coins = float('inf')
    for coin in coins:
        result = coin_change(amount - coin, coins)
        min_coins = min(min_coins, result + 1)
    return min_coins

# Note: Convert list to tuple for hashability
coin_change(11, tuple([1, 2, 5]))
```

Example 4: Graph with NetworkX

```
import networkx as nx

# Create graph
G = nx.DiGraph()
G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])

# Topological sort (Course Schedule)
try:
    order = list(nx.topological_sort(G))
    print(f'Valid order: {order}')
except nx.NetworkXError:
    print('Cycle detected!')

# Shortest path
```

```
path = nx.shortest_path(G, source=0, target=3)
print(f'Shortest path: {path}')
```

Part 6: Interview Best Practices

When to Use Libraries vs Manual Implementation

Scenario	Recommendation	Reason
Phone/initial screen	Ask if libraries allowed	Some allow, some don't
Take-home assignment	Use libraries liberally	Shows practical knowledge
Onsite whiteboard	Usually manual	Tests algorithm knowledge
Pair programming	Mix: use for utilities	Balance practicality and knowledge
LeetCode practice	Both: manual first, then lib	Learn both approaches
Production code	Always use libraries	Battle-tested, optimized

Key Takeaways

1. **Know what exists**: Awareness of library functions shows maturity
2. **Understand the algorithm**: Even if using a library, explain how it works
3. **Discuss tradeoffs**: Library vs manual, time vs space, readability vs performance
4. **Ask first**: 'Can I use Python's heapq module for this?' shows good communication
5. **Be prepared for both**: Practice manual implementation even if libraries exist
6. **Production mindset**: In real code, use libraries (don't reinvent the wheel)
7. **Time complexity still matters**: Know the complexity of library functions

Recommended Study Approach

1. **First pass**: Implement algorithm manually to understand it
2. **Second pass**: Refactor using appropriate library functions
3. **Compare**: Note differences in code length, readability, performance
4. **Memorize key libraries**: collections, heapq, bisect, itertools, functools
5. **Practice both ways**: Be fluent in manual and library approaches

---

This reference was generated as a companion to interview preparation scripts.
For the complete collection of interview problems and patterns, see the scripts/ directory.