

Python Data Structures & Algorithmic Complexity Cheat Sheet

Big-O time/space complexity, collections module, heap/bisect idioms, and amortized analysis.

EXECUTIVE SUMMARY

A production and technical-interview reference covering CPython internal memory layouts (dynamic over-allocated pointer arrays for list, open-addressing hash tables with compact indices for dict), Big-O time and space guarantees, and idiomatic standard-library patterns.

KEY EQUATIONS & INVARIANTS

* Amortized List Append Cost:

LaTeX: $T_{\text{amortized}}(n) = \frac{1}{n} \sum_{i=1}^n O(1) + \frac{O(n)}{n} = O(1)$

Note: CPython over-allocates by roughly 12.5% on resize, making n appends $O(n)$ total.

* Hash Table Load Factor Bound:

LaTeX: $\alpha = \frac{n}{m} \leq \frac{2}{3}$

Note: CPython resizes dict compact tables when load factor exceeds $2/3$ to keep probes $O(1)$.

* Binary Heap Parent / Child Indices:

LaTeX: $\text{parent}(i) = \lfloor (i-1)/2 \rfloor$, $\text{children}(i) = (2i+1, 2i+2)$

Note: 0-indexed array representation used by Python's `heapq` module (min-heap).

1. CPython Built-In Container Complexity Matrix

CPython's `list` is a contiguous array of `PyObject*` pointers (`ob_item`), not a linked list. Inserting or popping at index 0 shifts $n-1$ pointers ($O(n)$), whereas `collections.deque` uses a doubly-linked list of fixed-size 64-element blocks for $O(1)$ operations at both ends.

Structure | Access / Lookup | Insert / Append | Delete / Pop | Space

`list` (dynamic array) | $O(1)$ by index | $O(1)$ end / $O(n)$ head | $O(1)$ end / $O(n)$ mid | $O(n)$

`collections.deque` | $O(n)$ mid / $O(1)$ ends | $O(1)$ append / appendleft | $O(1)$ pop / popleft | $O(n)$

`dict` (compact hash) | $O(1)$ avg / $O(n)$ worst | $O(1)$ amortized | $O(1)$ amortized | $O(n)$

`set` / `frozenset` | $O(1)$ avg (`'x' in s`) | $O(1)$ amortized | $O(1)$ amortized | $O(n)$

`heapq` (binary min-heap) | $O(1)$ min (`h[0]`) | $O(\log n)$ `heappush` | $O(\log n)$

`heappop` | $O(n)$

[PYTHON] High-performance Sliding Window Monotonic Queue & Top-K Heap patterns

```
from collections import deque, Counter
import heapq
```

```
def sliding_window_max(nums: list[int], k: int) -> list[int]:
    """O(n) monotonic decreasing deque storing indices."""
    q: deque[int] = deque()
    out: list[int] = []
    for i, val in enumerate(nums):
        while q and nums[q[-1]] <= val:
            q.pop()
        q.append(i)
        if q[0] <= i - k:
            q.popleft()
```

```

        if i >= k - 1:
            out.append(nums[q[0]])
    return out

```

```

def top_k_frequent(words: list[str], k: int) -> list[str]:
    counts = Counter(words)
    return heapq.nsmallest(k, counts.keys(), key=lambda w: (-counts[w], w))

```

2. Binary Search (`bisect`) & Prefix Trees (Tries)

Use `bisect_left` to find the first index where `arr[i] >= target` and `bisect_right` for the first index where `arr[i] > target`. For string prefix matching, a dictionary-backed Trie provides $O(L)$ lookup independent of vocabulary cardinality N .

[PYTHON] Binary Search Invariants & Compact Dict Trie

```

import bisect

def count_in_range(sorted_arr: list[int], low: int, high: int) -> int:
    """Count elements x such that low <= x <= high in O(log n)."""
    left = bisect.bisect_left(sorted_arr, low)
    right = bisect.bisect_right(sorted_arr, high)
    return right - left

class PrefixTrie:
    def __init__(self) -> None:
        self.root: dict = {}

    def insert(self, word: str) -> None:
        node = self.root
        for ch in word:
            node = node.setdefault(ch, {})
        node['#'] = True

```

COMMON PRODUCTION & INTERVIEW PITFALLS

- Using `list.pop(0)` inside a BFS loop turns an $O(V + E)$ traversal into $O(V^2)$; always use `collections.deque.popleft()`.
- Initializing a 2D matrix with `[[0] * cols] * rows` aliases the same inner list across all rows; use `[[0] * cols for _ in range(rows)]`.
- Mutating a `dict` or `set` while iterating over it raises `RuntimeError: dictionary changed size during iteration`.

FREQUENTLY ASKED QUESTIONS

Q: Why is Python dict ordered since Python 3.7?

A: CPython splits the hash table into a compact `dk_indices` array (storing index positions) and a dense `dk_entries` array (appended in insertion order). This preserves insertion order while reducing memory footprint by ~20-25%.

Q: How do I use a max-heap in Python when heapq only implements a min-heap?

A: Negate numeric keys on push and pop (`heapq.heappush(h, -val)`), or on Python 3.14+ use the internal max-heap helpers, or wrap custom objects with an inverted `__lt__` comparator.