TimeComplexity

This page documents the time-complexity (aka "Big O" or "Big Oh") of various operations in current CPython. Other Python implementations (or older or still-under development versions of CPython) may have slightly different performance characteristics. However, it is generally safe to assume that they are not slower by more than a factor of O(log n).

Generally, 'n' is the number of elements currently in the container. 'k' is either the value of a parameter or the number of elements in the parameter.

list

The Average Case assumes parameters generated uniformly at random.

Internally, a list is represented as an array; the largest costs come from growing beyond the current allocation size (because everything must move), or from inserting or deleting somewhere near the beginning (because everything after that must move). If you need to add/remove at both ends, consider using a collections.deque instead.

Operation	**Average Case**	**Amortized Worst Case**
Copy	O(n)	O(n)
Append[1]	O(1)	O(1)
Pop last	O(1)	O(1)
Pop intermediate[2]	O(n)	O(n)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend[1]	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)	O(nk)
x in s	O(n)	
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

collections.deque

A deque (double-ended queue) is represented internally as a doubly linked list. (Well, a list of arrays rather than objects, for greater efficiency.) Both ends are accessible, but even looking at the middle is slow, and adding to or ends are accessible, but even looking at the middle is slow, and adding to or removing from the middle is slower still.

Operation	**Average Case**	**Amortized Worst Case**
Copy	O(n)	O(n)
append	O(1)	O(1)
appendleft	O(1)	O(1)
pop	O(1)	O(1)
popleft	O(1)	O(1)
extend	O(k)	O(k)
extendleft	O(k)	O(k)
rotate	O(k)	O(k)
remove	O(n)	O(n)

set

See dict -- the implementation is intentionally very similar.

Operation	**Average case**	**Worst Case**	**note**
x in s	O(1)	O(n)	
Union s|t	O(len(s)+len(t))		
Intersection s&t	O(min(len(s), len(t))	O(len(s) * len(t))	repla "min with "ma if t is not a set
Multiple intersection s1&s2&..&sn		(n-1)*O(l) where l is max(len(s1),..,len(sn))	
Difference s-t	O(len(s))		
s.difference_update(t)	O(len(t))		
Symmetric Difference s^t	O(len(s))	O(len(s) * len(t))	
s.symmetric_difference_update(t)	O(len(t))	O(len(t) * len(s))	

- As seen in the source code the complexities for set difference s-t or s.difference(t) (`set_difference()`) and in-place set difference s.difference_update(t) (`set_difference_update_internal()`) are different! The first one is O(len(s)) (for every element in s add it to the new set, if not in t). The second one is O(len(t)) (for every element in t remove it from s). So care must be taken as to which is preferred, depending on which one is the longest set and whether a new set is needed.

- To perform set operations like s-t, both s and t need to be sets. However you can do the method equivalents even if t is any iterable, for example s.difference(l), where l is a list.

dict

The Average Case times listed for dict objects assume that the hash function for the objects is sufficiently robust to make collisions uncommon. The Average Case assumes the keys used in parameters are selected uniformly at random from the set of all keys.

Note that there is a fast-path for dicts that (in practice) only deal with str keys; this doesn't affect the algorithmic complexity, but it can significantly affect the constant factors: how quickly a typical program finishes.

Operation	**Average Case**	**Amortized Worst Case**
k in d	O(1)	O(n)
Copy[3]	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item[1]	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration[3]	O(n)	O(n)

Notes

[1] = These operations rely on the "Amortized" part of "Amortized Worst Case". Individual actions may take surprisingly long, depending on the history of the container.

[2] = Popping the intermediate element at index **k** from a list of size **n** shifts all elements *after* **k** by one slot to the left using memmove. **n - k** elements have to be moved, so the operation is **O(n - k)**. The best case is popping the second to last element, which necessitates one move, the worst case is popping the first element, which involves **n - 1** moves. The average case for an average value of **k** is popping the element the middle of the list, which

takes **O(n/2) = O(n)** operations.

[3] = For these operations, the worst case *n* is the maximum size the container ever achieved, rather than just the current size. For example, if N objects are added to a dictionary, then N-1 are deleted, the dictionary will still be sized for N objects (at least) until another insertion is made.