

Searching, sorting & efficiency

Python Reference

Linear & binary search

A search finds where a value is. Linear search checks each item in turn, so it works on any list.

```
def linear_search(items, target):
    for i in range(len(items)):
        if items[i] == target:
            return i
    return -1    # not found

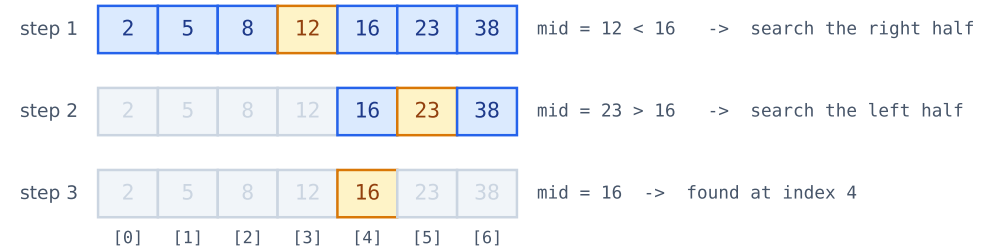
print(linear_search([4, 8, 2, 9], 2))    # 2
```

Binary search is much faster but needs a **sorted** list. It halves the range each step.

```
def binary_search(items, target):
    lo, hi = 0, len(items) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if items[mid] == target:
            return mid
        elif items[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1

print(binary_search([1, 3, 5, 7, 9], 7))    # 3
```

binary search for 16 (the array must be sorted)



binary search: halve the range each step · $O(\log n)$

Binary search halves the range each step — $O(\log n)$ on a sorted list

Sorting (bubble & insertion)

To sort is to put items in order. Bubble sort repeatedly swaps neighbours that are in the wrong order.

```
def bubble_sort(a):
    a = a[:]    # work on a copy
    for i in range(len(a)):
        for j in range(len(a) - 1 - i):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
    return a

print(bubble_sort([5, 2, 4, 1]))    # [1, 2, 4, 5]
```

Insertion sort builds a sorted part one item at a time, sliding each new item back into its place:

```
def insertion_sort(a):
    a = a[:]                # work on a copy
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1
        while j >= 0 and a[j] > key:  # shift bigger values right
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key          # drop key into the gap
    return a

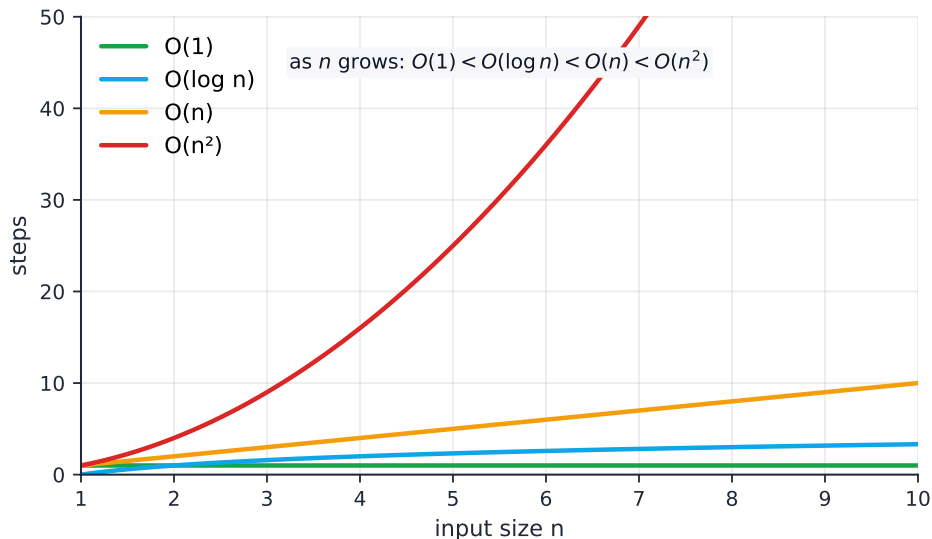
print(insertion_sort([5, 2, 4, 1]))  # [1, 2, 4, 5]
```

- In real code, use Python's built-in `sorted()`:

```
print(sorted([5, 2, 4, 1]))  # [1, 2, 4, 5]
```

Algorithmic efficiency

Efficiency asks how the work grows as the input grows. We describe it with Big-O O .



How the number of steps grows with input size for common complexities

Big-O	Name	Example
$O(1)$	constant	look up a dict key
$O(\log n)$	logarithmic	binary search
$O(n)$	linear	linear search
$O(n^2)$	quadratic	bubble sort

```
def steps(n):                # how many steps a linear scan takes
    count = 0
    for i in range(n):
        count = count + 1
    return count

print(steps(100))           # 100 -> O(n)
```

Randomness & simulation

The `random` module makes random numbers. Use a seed to make results repeatable. A simulation runs many random trials to estimate an answer.

```
import random
random.seed(0)
rolls = [random.randint(1, 6) for _ in range(1000)]
print(rolls.count(6))  # about 1/6 of 1000
```

Common mistakes

- Binary search only works on a **sorted** list.
- Big-O tells you how the time GROWS, not the exact time; an $O(n^2)$ method beats $O(n)$ only for tiny inputs.
- Bubble sort is $O(n^2)$ — fine for learning, but slow on large lists.