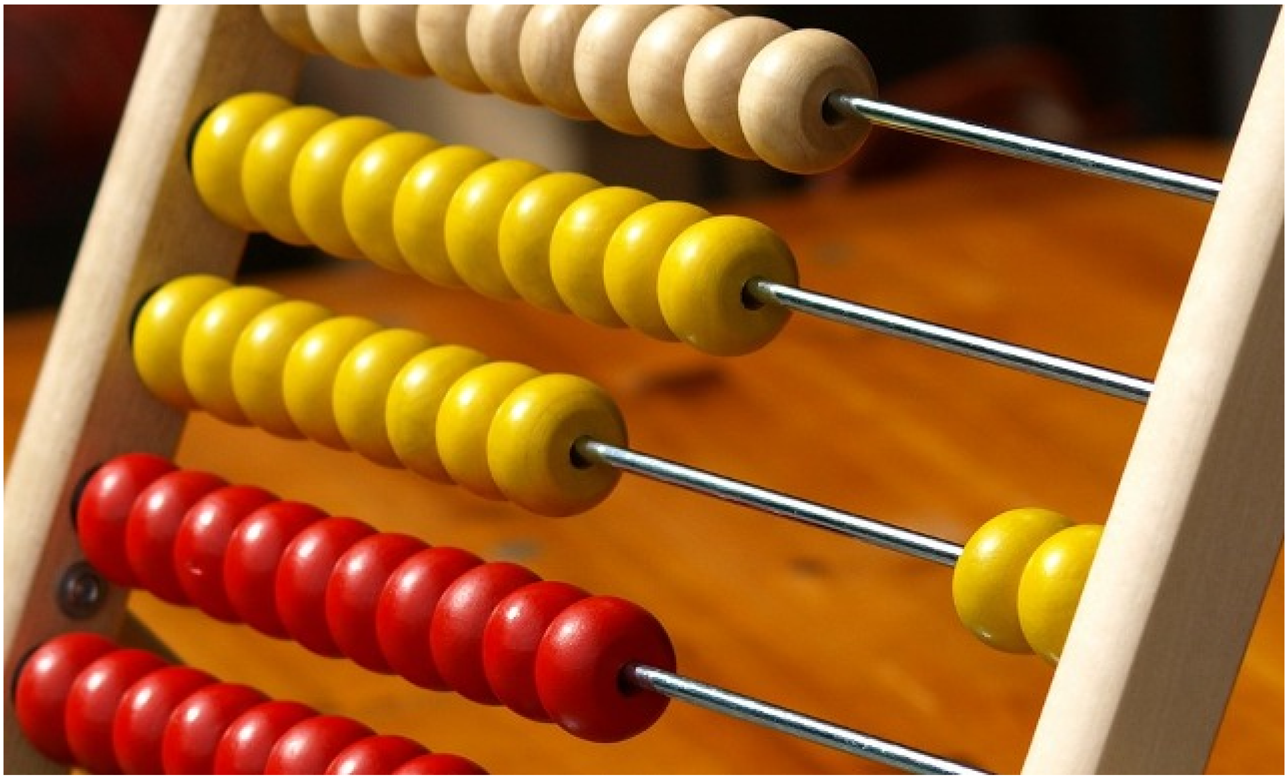
How to analyze time complexity: Count your steps

yourbasic.org

Time complexity estimates the time to run an algorithm. It's calculated by counting elementary operations.



Example (iterative algorithm)

What’s the running time of the following algorithm?

```
// Compute the maximum element in the array a.
Algorithm max(a):
    max ← a[0]
    for i = 1 to len(a)-1
        if a[i] > max
            max ← a[i]
    return max
```

The answer depends on factors such as input, programming language and runtime, coding skill, compiler, operating system, and hardware.

We often want to reason about **execution time** in a way that depends only on the **algorithm** and its **input**. This can be achieved by choosing an **elementary operation**, which the algorithm performs repeatedly, and define the **time complexity** $T(n)$ as the number of such operations the algorithm performs given an array of length n.

For the algorithm above we can choose the comparison `a[i] > max` as an elementary operation.

1. This captures the running time of the algorithm well, since comparisons dominate all other operations in this particular algorithm.
2. Also, the time to perform a comparison is constant: it doesn't depend on the size of `a`.

The time complexity, measured in the number of comparisons, then becomes $T(n) = n - 1$.

In general, an **elementary operation** must have two properties:

1. There can't be any other operations that are performed more frequently as the size of the input grows.
2. The time to execute an elementary operation must be constant: it mustn't increase as the size of the input grows. This is known as unit cost.

Worst-case time complexity

Consider this algorithm.

```
// Tell whether the array a contains x.
Algorithm contains(a, x):
    for i = 0 to len(a)-1
        if x == a[i]
            return true
    return false
```

The comparison `x == a[i]` can be used as an elementary operation in this case. However, for this algorithm the number of comparisons depends not only on the number of elements, n, in the array but also on the value of `x` and the values in `a`:

- If `x` isn't found in `a` the algorithm makes n comparisons,
- but if `x` equals `a[0]` there is only one comparison.

Because of this, we often choose to study **worst-case** time complexity:

- Let $T_1(n)$, $T_2(n)$, … be the execution times for all possible inputs of size n.
- The worst-case time complexity $W(n)$ is then defined as $W(n) = \max(T_1(n), T_2(n), \ldots)$.

The worst-case time complexity for the `contains` algorithm thus becomes $W(n) = n$.

Worst-case time complexity gives an upper bound on time requirements and is often easy to compute. The drawback is that it's often overly pessimistic.

> See Time complexity of array/list operations for a detailed look at the performance of basic array operations.

Average-case time complexity

Average-case time complexity is a less common measure:

- Let $T_1(n)$, $T_2(n)$, … be the execution times for all possible inputs of size n, and let $P_1(n)$, $P_2(n)$, … be the probabilities of these inputs.
- The average-case time complexity is then defined as $P_1(n)T_1(n) + P_2(n)T_2(n) + \ldots$

Average-case time is often harder to compute, and it also requires knowledge of how the input is distributed.

Quadratic time complexity

Finally, we'll look at an algorithm with poor time complexity.

```
// Reverse the order of the elements in the array a.
Algorithm reverse(a):
    for i = 1 to len(a)-1
        x ← a[i]
        for j = i downto 1
            a[j] ← a[j-1]
        a[0] ← x
```

We choose the assignment `a[j] ← a[j-1]` as elementary operation. Updating an element in an array is a constant-time operation, and the assignment dominates the cost of the algorithm.

The number of elementary operations is fully determined by the input size n. In fact, the outer for loop is executed n - 1 times. The time complexity therefore becomes

$W(n) = 1 + 2 + \ldots + (n - 1) = n(n - 1)/2 = n^2/2 - n/2$.

The quadratic term dominates for large n, and we therefore say that this algorithm has **quadratic** time complexity. This means that the algorithm **scales poorly** and can be used **only for small input**: to reverse the elements of an array with 10,000 elements, the algorithm will perform about 50,000,000 assignments.

In this case it's easy to find an algorithm with **linear** time complexity.

```
Algorithm reverse(a):
    for i = 0 to n/2
        swap a[i] and a[n-i-1]
```

This is a **huge improvement** over the previous algorithm: an array with 10,000 elements can now be reversed with only 5,000 swaps, i.e. 10,000 assignments. That's roughly a 5,000-fold speed improvement, and the improvement keeps growing as the the input gets larger.

It's common to use Big O notation when talking about time complexity. We could then say that the time complexity of the first algorithm is $\Theta(n^2)$, and that the improved algorithm has $\Theta(n)$ time complexity.