# 3 Algorithms and complexity

An algorithm is a complete and unambiguous specification of a procedure to solve a problem, expressed in a well-defined formal language. Sometimes people liken an algorithm with a cake recipe, but that is perhaps an ill-chosen metaphor, because cake recipes are often incomplete, ambiguous, and expressed in idiomatic and informal jargon. For example, in a cake recipe one might encounter instructions such as *season to taste* or *bake until golden*. In addition to being unequivocal, an algorithm must also begin with well defined initial conditions and be guaranteed to produce an output through a finite number of steps.

### *3.0.1 High level and implementation description*

When we describe an algorithm we often have two competing goals:

1. To explain the general idea, motivation, approach and method used in the algorithm to solve the particular problem at hand.

2. To provide sufficient detail to allow others to implement the algorithm and reproduce its results.

In a *high level description* of an algorithm, we describe the algorithm in normal language and ignore implementation details that are not necessary to understand the general approach. In an *implementation description*, we spell out exactly which actions must be performed and detail the initial conditions etc.

An implementation description is often specified in a formal language, such as (pseudo) computer code or in the form of a flowchart.

Example 3.1 Finding the smallest of three numbers

Let us say that we have three numbers, $a$, $b$, and $c$, and we are interested in finding the smallest number of the three. While we may say that the previous sentence is a high level description of the *problem*, it does not describe an algorithm. Let us formulate a high level description of an algorithm that can solve the problem:

> We go through each of the numbers and at each step we keep track of the smallest number seen so far. Once we have gone through all three numbers, the smallest number we have seen will be the smallest number of the three.

While this description characterizes the general idea, it leaves out many details. Next, we can spell out the implementation details, here in the form of a Python program:

```
def smallest_of_three(a, b, c):
    """Returns the smallest number amongst a, b, and c"""
   smallest = a
   if b < smallest:
      smallest = b
   if c < smallest:
      smallest = c
   return(smallest)
```

### 3.0.2 *Comparing algorithms*

There are usually many ways to solve a problem, so there might be many different algorithms that give the same result, but differ in terms of other properties. The two most important properties of algorithms are:

*Efficiency* How efficient is the algorithm in terms of its computational complexity and use of resources?

*Simplicity* How simple is the algorithm in terms of comprehensibility and amenability to be implemented on a computer?

Example 3.2 Multiplication of complex numbers

We know that two complex numbers can be multiplied together according to the following rule:

$$(a + ib) \cdot (c + id) = (ac - bd) + i(bc + ad).$$

Thus, the most simple and direct way to implement this would be something like the following.

```
def complex_multiplication(x, y):
    """Multiplies the complex numbers x and y"""
    (a,b), (c,d) = x, y
    real = a*c - b*d
    imag = b*c + a*d
    return((real, imag))
```

This algorithm requires four multiplications and two additions/subtractions. However, another algorithm that produces the exact same result was invented by Carl Friedrich Gauss, requiring only three multiplications and five additions/subtractions.

```
def gauss_complex_multiplication(x, y):
    """Multiplies the complex numbers x and y"""
    (a,b), (c,d) = x, y
    k1 = c * (a+b)
    k2 = a * (d-c)
    k3 = b * (c+d)
    real = k1 - k3
    imag = k1 + k2
    return((real, imag))
```

In the good old days, multiplication was a much more expensive and time-consuming computation compared to addition and subtraction, both when it was carried out by hand and on a computer. For that reason, Gauss' algorithm is more efficient but certainly less easy to understand.

## 3.1 Algorithmic complexity

In the analysis of algorithms, *algorithmic complexity* is an approach to classifying algorithms according to their computational demands. The *time complexity* is a measure of how fast the algorithm will run on a computer. Since not all computers are the same, the time complexity is most often measured by the required number of elementary operations, such as multiplications and additions etc. Similarly, the *storage complexity* of an algorithm measures how much memory the computer needs in order to run the algorithm. In the following we will focus on time complexity, but most of the discussion applies to analysis of storage complexity as well.

### 3.1.1 *Dependence on input*

For many algorithms, the running time will depend on the input: If the input to some algorithm is a list of numbers, the running time might depend both on the length of the list (how many numbers the algorithm needs to process), but also on the specific values on that input list.

> Example 3.3 Linear search
>
> Let us consider an algorithm designed to identify whether or not a particular number `a` occurs somewhere in a list of numbers `x`. Such an algorithm might be implemented by the following Python code:
>
> ```
> def contains_value(x, a):
>     """Returns True only if list x contains value a"""
>     for val in x:
>         if val==a:
>             return(True)
>     return(False)
> ```
>
> The algorithm will be very fast in case there is a value `a` somewhere in the beginning of the list. Once the algorithm has found an `a`, it can return in the affirmative, whereas if there is no `a` anywhere on the list, the algorithm must visit all the values on the list `x` before it can return its negative result.

### 3.1.2 *Best, average, and worst case*

To take into account that the complexity of an algorithm can depend on the value of the input, we can for example specify the best case, average case, or worst case complexity.

*The best case complexity* is the minimal complexity attainable given the most favorable input.

*The worst case complexity* is the maximal complexity attainable given the most unfavorable input.

*The average case complexity* is the complexity attained for some reasonable definition of an average use case. The average case might be analyzed through a set of typical inputs, or computed as the average complexity over all possible inputs.

We can then quantify the complexity by a function $T(n)$ that counts the number of operations required for the algorithm to process an input of length $n$.

Example 3.4 Best, worst, and average case complexity

Consider again the algorithm in Example 3.3 that determines whether or not a value `a` is somewhere on a list.

*In the best case* the value `a` occurs as the first element on the list, and the algorithm will finish after making a single variable comparison. We can then say that the best case time complexity is $T(n) = 1$.

*In the worst case* the value `a` does not occur on the list, and the algorithm will finish only after comparing `a` to all input values. This requires $n$ comparisons, where $n$ is the length of the input list. We say that the worst case time complexity is $T(n) = n$.

*In the average case* we would need a further assumption about how input lists might look like in a typical use case. For example, we might assume that about half the time, the list does not contain `a`, and the other half of the time it contains a single `a` at some random position. In the fist case, it requires $n$ comparisons, whereas in the second case it requires $\frac{1}{2}n$ comparisons on average. In total, under these assumptions the average case would require $\frac{1}{2}(n + \frac{1}{2}n) = \frac{3}{4}n$ comparisons. We can then say that the average case time complexity is $T(n) = \frac{3}{4}n$.

### 3.1.3 *Big O-notation*

Big O-notation is useful for analyzing how much time or how many operations it takes to compute an algorithm. It is an asymptotic notation, that expresses how the computational complexity grows as the input grows in size. Formally, the big O-notation can be defined as follows.

**Definition 3.1 Big O-notation**

The computational complexity is

$$T(n) \in O\big(f(n)\big)$$

if and only if there exists a constant $c$ such that $T(n) < c \cdot f(n)$ for all $n > n_0$. We say $f(n)$ is an asymptotic upper bound for $T(n)$.

Here, $T(n)$ is the time complexity that measures the number of computational operations required to run the algorithm. The big O notation focuses only on how the complexity grows with $n$: It ignores any multiplicative constants and only considers the fastest growing term in $T(n)$.

**Example 3.5**

Let's say that we have determined that an algorithm requires

$$T(n) = 2n^2 + 10n + 50$$

computational operations to run. Now, let us determine the asymptotic complexity: We can verify that

$$T(n) < 6n^2,$$

for all $n > 5$ (see Figure 3.1) so the computational complexity is "big O of n squared",

$$T(n) \in O(n^2).$$

We can easily get to this conclusion simply by identifying the fastest growing term and ignoring any multiplicative constants.

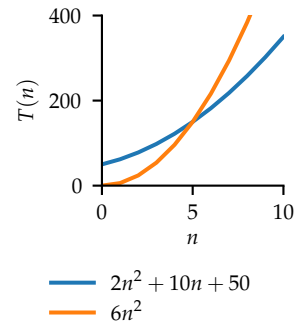


Figure 3.1: Example of running time complexity.

**Example 3.6 Big O for linear search**

Consider again the algorithm in Example 3.3 that determines whether or not a value a is somewhere on a list. In Example 3.4 we determined under certain assumptions that the average case complexity of the algorithm was

$$T(n) = \tfrac{3}{4}n$$

Considering Definition 3.1 we can see that $f(n) = n$ is an asymptotic upper bound for $T(n)$, and so we can say that the computational complexity is "big O of n",

$$T(n) \in O(n).$$

### Example 3.7 Duplicates on a list

The following algorithm goes through each unique pair of values in a list `x`, and determines whether or not the list contains one or more duplicates. As soon as the algorithm finds a duplicate, it will stop and return `True`. If the list does not contain any duplicates, the algorithm will return `False` after comparing all possible pairs of values.

```
def contains_duplicates(x):
    """Returns True if list x contains any duplicates"""
   for i in range(len(x)-1):
      for j in range(i+1, len(x)):
         if x[i] == x[j]:
            return(True)
   return(False)
```

If the list contains $n$ numbers, the total number of unique pairs to compare is $n(n-1)/2$ (can you show this?). Thus, the worst case computational complexity occurs in the situation where the list contains no duplicates. Counting the number of variable comparisons needed to run the algorithm, we end up with a worst case computational complexity of

$$T(n) = \frac{n(n-1)}{2} = \tfrac{1}{2}n^2 - \tfrac{1}{2}n.$$

We can see that $T(n)$ is upper bounded by $c \cdot n^2$ for all $n > n_0$ with $c = \frac{1}{2}$ and $n_0 = 0$. Therefore we can say that the computational complexity is "big O of $n$ squared",

$$T(n) \in O(n^2).$$

## Problems

1. Consider the algorithm for finding the smallest of three values described in Example 3.1. Counting the number of *variable assignments* and *variable comparisons*, what is the best case, average case, and worst case time complexity of the algorithm.

## Solutions

1. In the best case `a` is the smallest, so we only need to make the first variable assignment and compute the two comparisons, so $T = 3$. In the worst case `c` is smallest and `b` is smaller than `a`, so all code lines will be run and $T = 5$. In the average case, we can consider all six possible orderings of the three numbers: On average we have $T = 3.83$.