



# Algorithm Analysis

---

Michael Hahsler



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).






# What is an Algorithm?

## Definition

*“A clearly specified set of simple instructions to be followed to solve a problem.”*

## Important properties

- Correctness
- **Run time** - **time complexity**
- Memory requirement - space complexity



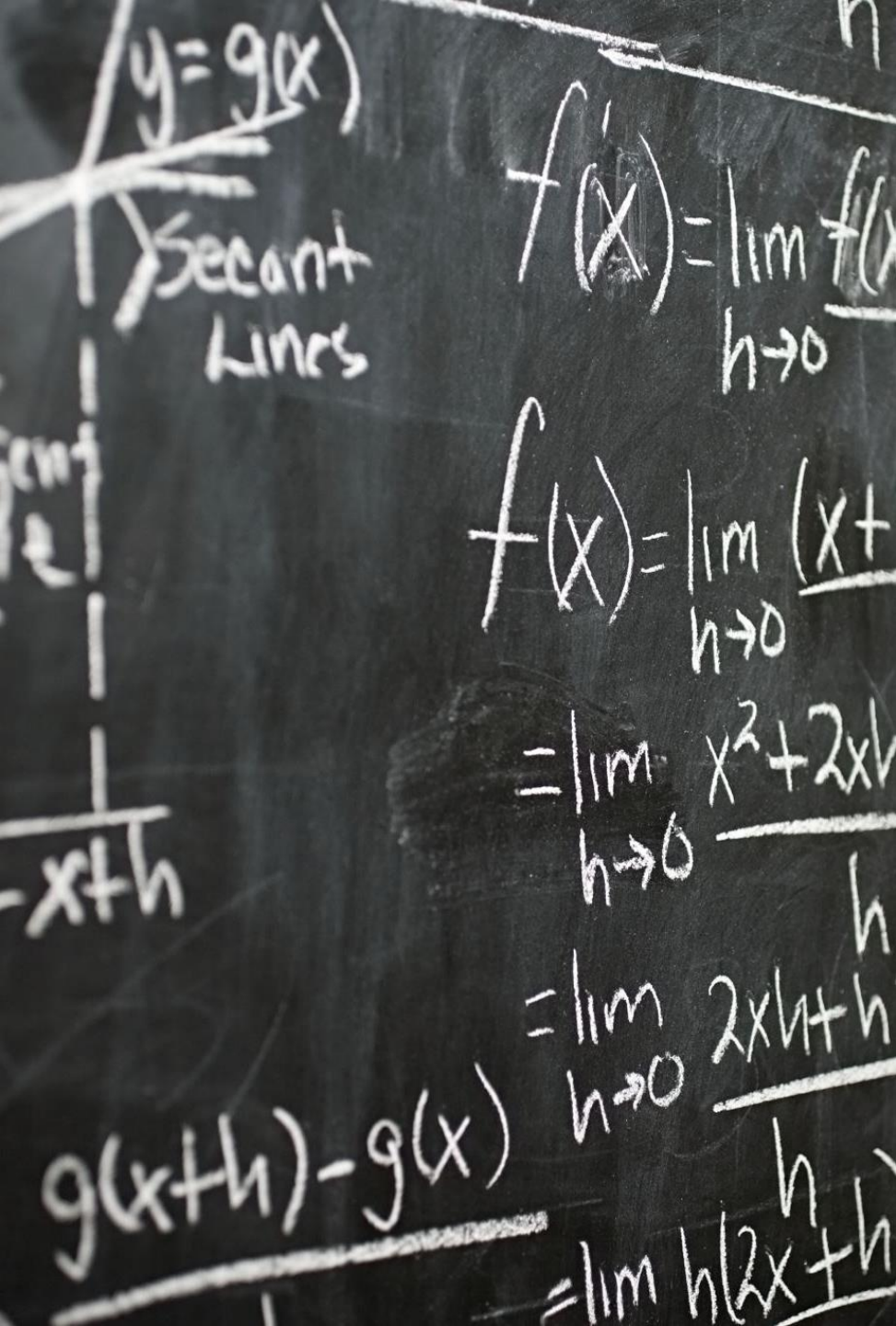
# How do we assess/compare time complexity?

---

Issue: Runtime and memory requirements are machine/programming language/compiler dependent!

## Approach

1. Assume that each instruction takes one time unit.
2. Define the problem size (typically called  $N$ ) and look at the relative growth rate when  $N$  becomes large.
3. We typically analyze the worst-case behavior.



# Mathematical Definitions

## Big-Oh

(upper bound = worst case, i.e. the growth is not more than  $f$ )

$T(N) = O(f(N))$  if there is a  $c$  for which  $T(N) \leq cf(N)$   
for  $N > n_0$

Most common

## Omega

(lower bound, i.e. the growth is not less than  $g$ )

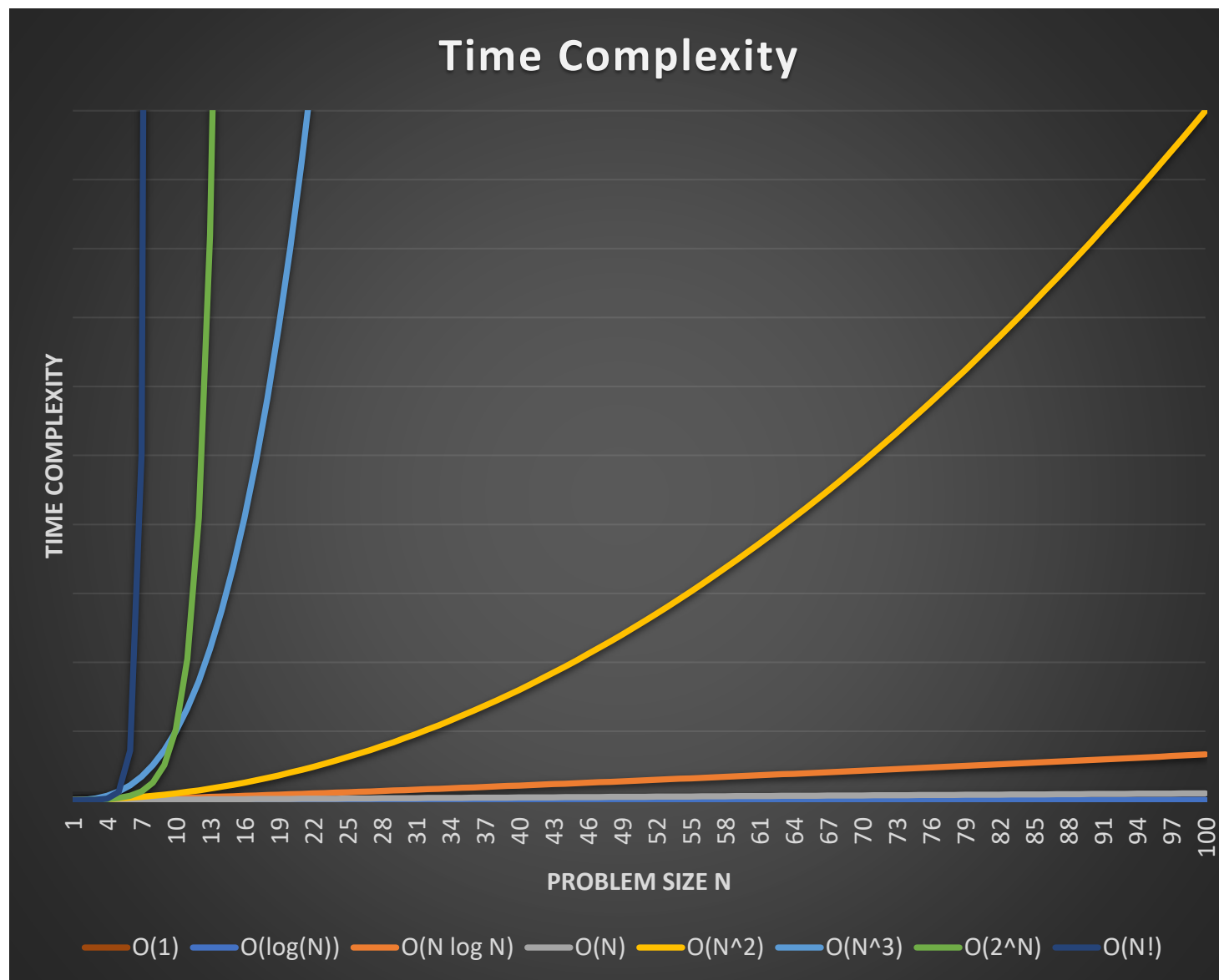
$T(N) = \Omega(g(N))$  if there is a  $c$  for which  $T(N) \geq cg(N)$   
for  $N > n_0$

## Theta

(exact growth rate is  $h$ )

$T(N) = \Theta(h(N))$  if  $T(N) = O(h(N))$  and  $T(N) = \Omega(h(N))$

Comparison  
of different  
complexity  
functions  $f$



# Some Rules

1. Ignore constants

$$f(N) = 5 + 3 N^2 \rightarrow O(N^2)$$

2. Only keep the highest degree since it grows the fastest with  $N$

$$f(N) = N^2 + N^3 \rightarrow O(N^3)$$

3. Sequence:  $O(f(N)) + O(g(N)) = O(\max[f(N), g(N)]) \rightarrow$  the worst part counts.

4. Nested loops:  $O(f(N)) * O(g(N)) = O(f(N)g(N)) \rightarrow$  do  $g(N)$   $f(N)$  times.

For code this means

- Only loops really count:  $k$  nested loops over  $N \rightarrow O(N^k)$
- Consecutive code blocks: only the most complex block counts.
- Conditional code blocks: count the most complex path (worst case)

# Some Useful Series

$$\sum_{i=0}^N s^i = 2^{N+1} - 1$$

$$\sum_{i=1}^N i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

$$\sum_{i=1}^N i^k \approx \frac{N^{k+1}}{|k+1|} \text{ for } k \neq -1$$

$$\sum_{i=1}^N \frac{1}{i} \approx \log_e N$$

# Example: Bubble Sort

```
// n is the length of the arr
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; ++i)

        // Last i elements are already
        // in place
        for (j = 0; j < n - i - 1; ++j)
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]);
}
```

Worst-case time complexity?

Worst-case space complexity?



# Example: Bubble Sort

```
// n is the length of the arr
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; ++i)

        // Last i elements are already
        // in place
        for (j = 0; j < n - i - 1; ++j)
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]);
}
```

Worst-case time complexity?

i = 1 : n - 1 iterations in j-loop

i = 2 : n - 2

i = 3 : n - 3

...

i = n - 2 : n - (n - 2) - 1 = 1

i = n - 1 : n - (n - 1) - 1 = 0

$$\sum_{i=0}^{n-1} i = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \rightarrow O(n^2)$$

Worst-case space complexity?

Only memory for i and j is needed  $\rightarrow O(1)$

# Example: Linear Search

```
template <typename Comparable>
int linearSearch(
    const vector<Comparable> & a,
    const Comparable & x )
{
    for(int i = 0; i < a.size(); ++i)
    {
        if( a[ i ] == x )
            return i;
    }
    return -1
}
```

Find the index of a value in a presorted array/vector.

Time complexity?

# Example: Linear Search

```
template <typename Comparable>
int linearSearch(
    const vector<Comparable> & a,
    const Comparable & x )
{
    for(int i = 0; i < a.size(); ++i)
    {
        if( a[ i ] == x )
            return i;
    }
    return -1
}
```

Time complexity?

1. Problem size  $n$  = size of  $a$  represented as high-low.
2. Worse case is if we do not find the element  $x$ . We have to iterate  $n$  times  $\rightarrow O(n)$

# Example: Binary Search

```
template <typename Comparable>
int binarySearch(
    const vector<Comparable> & a,
    const Comparable & x )
{
    int low = 0, high = a.size( ) - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else if( a[ mid ] > x )
            high = mid - 1;
        else
            return mid;    // Found
    }
    return -1
}
```

Find the index of a value in a presorted array/vector.

Time complexity?

# Example: Binary Search

```
template <typename Comparable>
int binarySearch( const vector<Comparable> & a,
                  const Comparable & x )
{
    int low = 0, high = a.size( ) - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else if( a[ mid ] > x )
            high = mid - 1;
        else
            return mid; // Found
    }
    return -1
}
```

## Time complexity?

1. Problem size  $n$  = size of  $a$  represented as high-low.
2. Worse case is if we do not find the element  $x$ .
3. In every iteration, high-low halves so we get:

$$n, \frac{n}{2}, \frac{n}{4}, \dots, 1$$

4. It takes  $k$  steps to get from  $N$  to 1:

$$\begin{aligned}\frac{n}{2^k} &= 1 \\ n &= 2^k \\ k &= \frac{\log_2(n)}{\log_2(2)}\end{aligned}$$

$$\rightarrow O(\log n)$$

**General rule:** If every iteration halves the problem size, then we have a logarithmic time complexity.

# Some Final Words on Complexity

---

- Better complexity does not mean faster! Complexity analysis ignores constants. **runtime**
- **Assumption** that each operation takes the same amount of time is very strong.
- Complexity analysis points to the part of the code that would benefit from algorithmic **optimization**.
- Complexity analysis looks at the **algorithm** and bad implementations (e.g., copying arrays unnecessarily) may lead to worse run time.
- **Big-Oh looks at the worst case.** Average case analysis is typically much harder to determine.
- Space complexity analysis looks at how the memory need grows with  $N$ .

