Algorithm Analysis

Michael Hahsler

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

What is an Algorithm?

Definition

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

Important properties

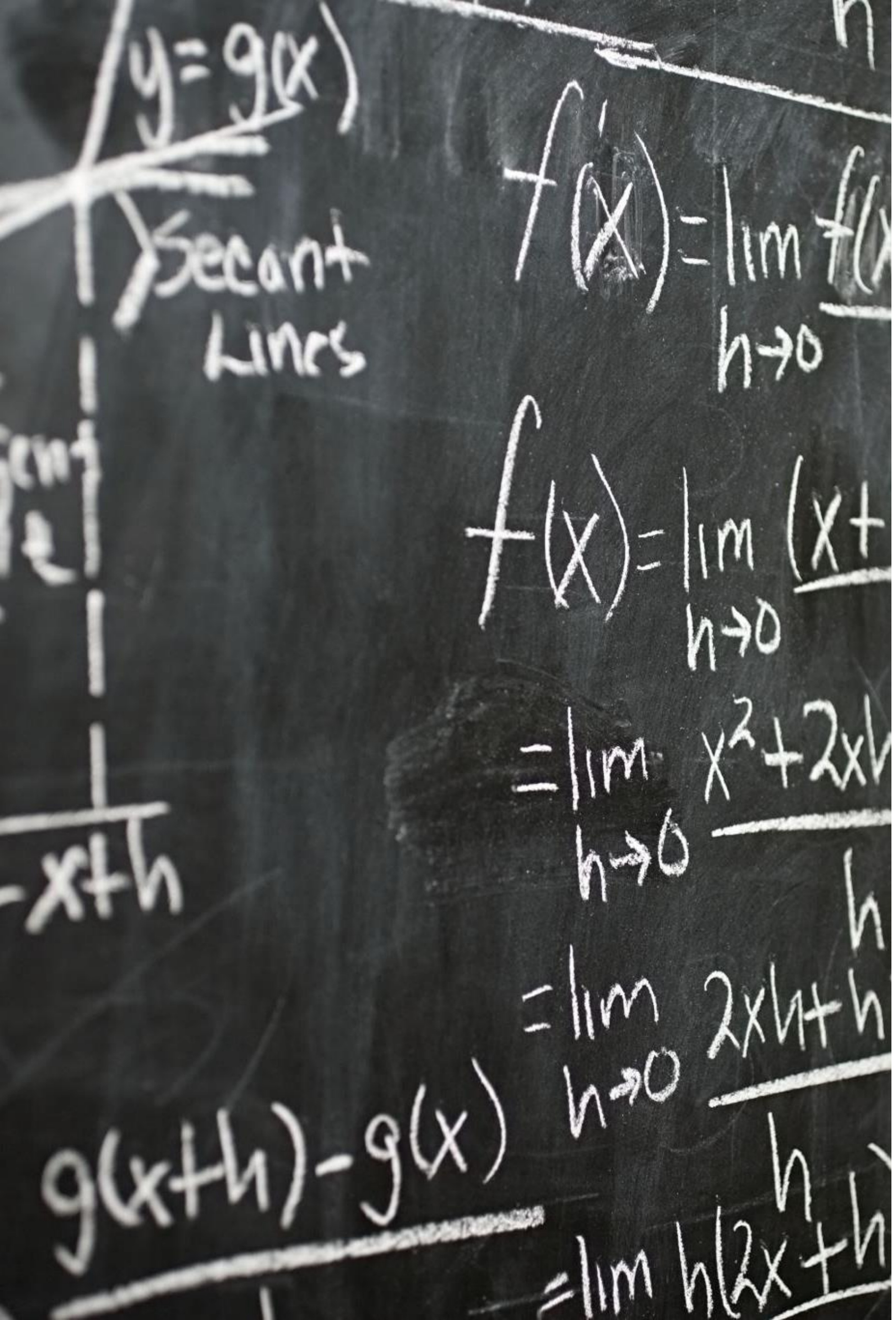
- Correctness
- **Run time** **- time complexity**
- Memory requirement - space complexity

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 in not more than f)

$\boldsymbol{T(N) = O\big(f(N)\big)}$ **if there is a** $\boldsymbol{c}$ **for which** $\boldsymbol{T(N) \le cf(N)}$
for $\boldsymbol{N > n_0}$

Most common

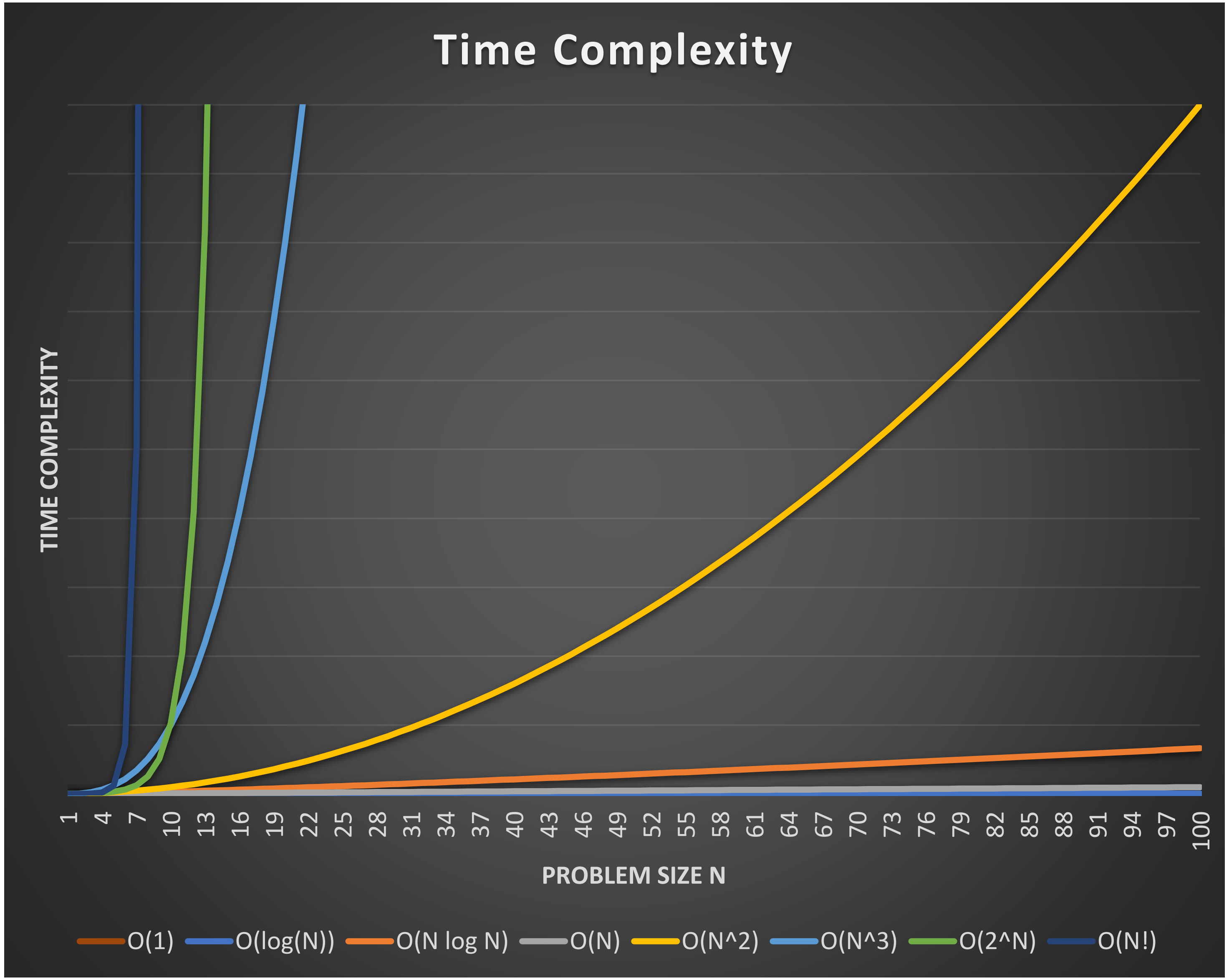
Omega
(lower bound, i.e. the growth in not less than g)

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

Theta
(exact growth rate is h)

$T(N) = \Theta\big(h(N)\big)$ if $T(N) = O\big(h(N)\big)$ and $\mathrm{T(N)} = \Omega\big(h(N)\big)$

Comparison of different complexity functions f



Some Rules

1. Ignore constants

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

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

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

Example

```
// 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?

Some Useful Series

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

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

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

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

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 way harder to determine. Also, it does not give a guarantee.
- Space complexity analysis looks at how the memory need grows with N.