

Complexity

Christian Sattler

2020-11-09

Complexity

★ What is computational complexity?

- Orders of growth
- Determining complexity
- “Big O” Notation

What is complexity?

Computational complexity of an algorithm:

- ★ How much resources does the algorithm use in relation to quantitative properties of its input?

What is complexity?

Computational complexity of an algorithm:

- ★ How much resources does the **algorithm** use in relation to quantitative properties of its input?

Algorithm: any well-defined procedure to solve a problem:

- ★ Sorting an array of objects: insertion sort, merge sort, quick sort, etc.
- ★ Adding an object to a dynamic array
- ★ Computing the first N digits of π
- ★ ...

What is complexity?

Computational complexity of an algorithm:

- ★ How much **resources** does the algorithm use in relation to quantitative properties of its input?

Resources:

- ★ Time (runtime): **time complexity** Most important
- ★ Space (memory, storage): **space complexity**
- ★ Other kinds of resources:
 - Number of comparisons (searching, sorting)
 - Number of arithmetic operations (e.g. matrix multiplication)

What is complexity?

Computational complexity of an algorithm:

- ★ How much resources does the algorithm use in relation to **quantitative properties** of its **input**?

Quantitative input properties:

- ★ Most often, some form of **size**:
 - Length of input string
 - Number of objects in a data structure (e.g. array)
- ★ Can be **multiple numbers**:
 - Numbers $|V|, |E|$ of vertices and edges in a graph (V, E)
 - Dimension and sparsity of a matrix
- ★ The input itself (e.g., n for “first n digits of π ”)

What is complexity?

Complexity is a **function** from input parameters to resource usage.

- Time complexity: runtime $T(n)$ as a function of input size n .

Different ways to account for variability:

★ **Worst-case** complexity

Most common, the default

- $T(n)$ is maximum runtime over all possible inputs of size n .

★ **Average-case** complexity

- $T(n)$ is average runtime over all possible inputs of size n .
- Requires a probability distribution over inputs (e.g. uniform).

★ **Amortized** complexity

- Average $T(n)$ over many sequential calls of the algorithm.
- Example: adding an element to a dynamic array
- Can be used with both worst-case and average-case.

Time complexity

Caveat. The exact runtime $T(n)$ of an algorithm depends...

- ★ ...theoretically: on the chosen model of computation,
- ★ ...in practice: on software/hardware details of the implementation.

Time complexity: model of computation

Caveat. The exact runtime $T(n)$ of an algorithm depends...

- ★ ...theoretically: on the chosen **model of computation**,
- ★ ...in practice: on software/hardware details of the implementation.

A popular model is the **RAM** (random access machine):

- ★ Idealized computer with infinite memory
- ★ Memory cells are addressed by and store natural numbers
- ★ Basic operations take **one time step**:
 - Arithmetic operations
 - Accessing a memory cell (including indirect addressing)
 - Loops and subroutine calls include “inner” costs
- ★ Good balance between simplicity and similarity to real computers

Time complexity: practical runtime

Caveat. The exact runtime $T(n)$ of an algorithm depends...

- ★ ...theoretically: on the chosen model of computation,
- ★ ...in practice: on **software/hardware details** of the implementation.

Numerous influences on runtime:

- ★ Compiler (translating algorithm to machine code)
- ★ CPU (instruction speeds, out of order execution),
- ★ memory architecture (sizes/levels of cache)
- ★ operating system managing processes

Extremely difficult to model precisely!

- ★ Runtime measurements usually not precisely reproducible.

Time complexity: conclusion

Caveat. The exact runtime $T(n)$ of an algorithm depends...

- ★ ...theoretically: on the chosen model of computation,
- ★ ...in practice: on software/hardware details of the implementation.

Conclusion.

- ★ Exact values of $T(n)$ not relevant.
- ★ $T(n)$ only well-defined up to constant factor.
 - This is called **order of growth** of $T(n)$ in n .
 - This is what we typically mean by **complexity**.
- ★ What matters is the **asymptotic behaviour** of $T(n)$ as n grows large.

$$\lim_{n \rightarrow \infty} \frac{T(n)}{Cf(n)} = 1 \quad \text{OR} \quad \lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = C$$

Handwritten notes: $T(n) \propto c f(n)$

What are asymptotes: →

Complexity

- What is computational complexity?
- ★ Orders of growth
- Determining complexity
- “Big O” Notation

Orders of growth

Common orders of growth (in n), by order:

- ★ 1 constant
- ★ $\log(n)$ logarithmic
- ★ n linear
- ★ $n \log(n)$ linearithmic
- ★ n^2 quadratic
- ★ 2^n exponential



Orders of growth

Common orders of growth (in n), by order, with typical algorithms:

★ 1	constant	array lookup, hash map lookup
★ $\log(n)$	logarithmic	binary search, effective ops. on data structures
★ n	linear	copying, looping over input
★ $n \log(n)$	linearithmic	efficient sorting algorithms
★ n^2	quadratic	insertion sort, double loop over input
★ 2^n	exponential	intractable problems, computing all subsets of a set

Orders of growth: runtime illustrated

For illustration purposes, suppose that:

- ★ Time complexity $T(n)$ of algorithm is one of below functions,
- ★ unit of runtime is 1 nanosecond ($1 \text{ ns} = 10^{-9} \text{ s}$).

Runtime of algorithm for various input sizes:

n	1	$\log(n)$	n	$n \log(n)$	n^2	2^n
10	1 ns	2 ns	10 ns	23 ns	100 ns	1 μ s
20	1 ns	3 ns	20 ns	60 ns	400 ns	1 ms
100	1 ns	5 ns	100 ns	460 ns	10 μ s	10^{14} years
1000	1 ns	7 ns	1 μ s	7 μ s	1 ms	-
10^6	1 ns	14 ns	1 ms	14 ms	17 min	-
10^9	1 ns	21 ns	1 s	21 s	31 years	-

Complexity

- What is computational complexity?
 - Orders of growth
- ★ Determining complexity
- “Big O” Notation

Determining complexity

For definiteness, we stick to the time complexity $T(n)$ of an algorithm with input parametrized by some notion of size n .

We will discuss two approaches to determining complexity:

★ Empirical approach

- Buy a computer, run the algorithm for a range of input sizes.
- Measure runtime (or memory usage), fit model.

★ Theoretical approach (rigorous)

- Compute $T(n)$ in RAM (random access machine) model.
- Derive complexity as order of growth of $T(n)$.

Empirical approach

- ★ Measure runtime $T(n)$ for $n = 1, a, a^2, \dots$ (geometric series) for some $a > 1$.
 - Use $n = 1, 2, 3, \dots$ to quickly detect exponential complexity.
 - For instantaneous runs, repeat and average $T(n)$ to counter noise.
- ★ Try to see the growth behaviour in the resulting runtimes.
 - Takes practice!
 - Heuristic: $T(a^{k+1})/T(a^k) \sim a^d$ if $T(n)$ has order of growth n^d .
 $(\log T(a^{k+1}) - \log T(a^k)) / \log a \sim d$ reveals exponent.
- ★ Fit $T(n) \sim C f(n)$ against data series for chosen model $f(n) = 1, \log n, n, n^d, 2^n$
 - Least squares regression
 - Alternative: fit $\log T(n) \sim C + \log f(n)$.
 - Pedestrian alt.: does $T(n)/f(n)$ stay constant (or within fixed interval)?

Empirical approach

Downsides:

- ★ Measures average-case complexity, not worst-time complexity.
- ★ Measurements for not too big n severely dependent on computer architecture (cache sizes).
- ★ Difficult to detect logarithmic factors ($\log n$).
- ★ Has difficulty detecting “non-smooth” order of growth (fluctuating constant factor).
- ★ You need a computer!

Theoretical approach

Calculate (an upper bound) for $T(n)$ using the RAM model.

💡 “**Cheat**”: We work with higher level of “pseudocode” (close to Java).

- ★ Count basic instructions at level of pseudocode.
- ★ Features of pseudocode such as richer types or garbage collection can be implemented in the RAM model
 - Take on faith (or verify): this adds only a constant factor!

Theoretical approach: example

Example algorithm: searching for a number y in an array xs .

★ n : length of given array xs

★ Algorithm:

```
bool contains(int[] xs, int y):
    bool found = false;
    for i from 0 to xs.length:
        If xs[i] == y
            found = true
    return found
```

1	
1 + n	
n • 1	
k • 1	

where k is the number of times y appears in xs .

★ $T(N) = 1 + (1 + n) + n + k = 2 + 2n + k$

★ In the worst case, $k = n$, so $T(N) = 2 + 3n$.

Conclusion: The order of growth is linear.

Theoretical approach: problems

Downsides:

- ★ This was only a simple algorithm. The final expression for $T(n)$ will get more and more as more summands appear.
- ★ Even worse: depending on the conditionals in the algorithm, $T(n)$ might not have a simple, closed form expression.

We will learn a way around both issues in the next section.

Complexity

- What is computational complexity?
- Orders of growth
- Determining complexity
- ★ “Big O” Notation

“Big O” notation

We need a formalism for **comparing and calculating orders of growths.**

Mathematicians to the rescue: 

- ★ “Big O” notation / O-notation
- ★ $f(n) = O(g(n))$: “ $f(n)$ has order of growth (at most) $g(n)$ ”

Definition. Given functions $f(n)$ and $g(n)$, we write $f(n) = O(g(n))$ if there are constants C and n_0 such that $f(n) \leq C g(n)$ for $n \geq n_0$.

- ★ $f(n)$ is “eventually” (after n_0) bounded by $C g(n)$.

If you have trouble with O-Notation, always go back to this definition!

Orders of growth

Now we can make our previous classification more precise!

Common orders of growth (in n), by order, with typical algorithms:

★ 1	constant	array lookup, hash map lookup
★ $\log(n)$	logarithmic	binary search, effective ops. on data structures
★ n	linear	copying, looping over input
★ $n \log(n)$	linearithmic	efficient sorting algorithms
★ n^2	quadratic	insertion sort, double loop over input
★ 2^n	exponential	intractable problems, computing all subsets of a set

Orders of growth: complexity classes

Now we can make our previous classification more precise!

Common **complexity classes**, by order, with typical algorithms:

★ $O(1)$	constant	array lookup, hash map lookup
★ $O(\log(n))$	logarithmic	binary search, effective ops. on data structures
★ $O(n)$	linear	copying, looping over input
★ $O(n \log(n))$	linearithmic	efficient sorting algorithms
★ $O(n^2)$	quadratic	insertion sort, double loop over input
★ $O(2^n)$	exponential	intractable problems, computing all subsets of a set

“Big O” notation: examples

Example. $13n + 37 = O(n)$.

Proof.

- ★ We need C and n_0 such that $13n + 37 \leq Cn$ for $n \geq n_0$.
- ★ Pick $C = 14$ and $n_0 = 37$.
- ★ Then $13n + 37 \leq 13n + n = 14n = Cn$ for $n \geq n_0$.

“Big O” notation: examples

Example. The statement $n = O(1)$ is false.

Proof.

- ★ Suppose $n = O(1)$. Then there are C and n_0 such that $n \leq C$ for $n \geq n_0$.
- ★ Pick an n larger than C and n_0 .
- ★ Then $C < n \leq C$, i.e. $C < C$, a falsehood.

“Big O” notation: examples

Example. $K = O(1)$.

Proof.

- ★ We need C and n_0 such that $K \leq C \times 1$ for $n \geq n_0$.
- ★ This holds for $C = K$ and n_0 arbitrary.

“Big O” notation: basic properties

Transitivity. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

Proof.

★ We are given:

- C, n_0 such that $f(n) \leq C g(n)$ for $n \geq n_0$
- C, n_0' such that $g(n) \leq C' h(n)$ for $n \geq n_0'$

★ Take $C'' = C C'$ and $n_0'' = \max(n_0, n_0')$.

★ Then $f(n) \leq C g(n) \leq C (C' h(n)) = C'' h(n)$ for $n \geq n_0''$.

This allows us to simplify the order of growth of expressions **step by step**.

“Big O” notation: basic properties

The O-formalism is compatible with basic arithmetic operations.

Suppose $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Then:

- ★ $f_1(n) f_2(n) = O(g_1(n) g_2(n))$
- ★ $f_1(n) + f_2(n) = O(g_1(n) + g_2(n)) = O(\max(g_1(n), g_2(n)))$

(Proofs: exercise!)

This tells us how to **add** and **multiply** orders of growths.

“Big O” notation: basic properties

Consequences of these properties:

- ★ Constant factors can be ignored: $K f(n) = O(f(n))$.
- ★ $n^a + n^b = O(n^{\max(a, b)})$ for $a, b \geq 0$.

(Try to deduce this from the two previous slides!)

With this: the order of growth of a **polynomial** is determined by its leading exponent.

- ★ $5n^3 + 12n^2 - 5n + 3 = O(n^3)$
- ★ $n^2 + 100000n = O(n^2)$
- ★ $n^k + n^{k-1} + \dots + n + 1 = O(n^k)$

“Big O” notation for runtime complexity

O-notation greatly simplifies the runtime calculations

- ★ **Sequence** of k (constant) statements

$A_1 \ O(f_1(n)), \dots, A_k \ O(f_k(n))$

has complexity $O(\max(f_1(n), \dots, f_n(n)))$.

- ★ **Loop** with $O(f(n))$ iterations and **non-empty body** complexity $O(g(n))$
has complexity $O(f(n) \cdot g(n))$.

Example. Program from before: $T(n) = O(1) + O(n) \cdot O(1) + O(1) = O(n)$.

```
bool contains(int[] xs, int y):
    bool found = false          0(1)
    for i from 0 to xs.length   0(n) iterations
        if xs[i] == y:
            found = true }      0(1)
    return found                0(1)
```