Introduction to Time and Space Complexity

The *time complexity* of an algorithm quantifies the amount of time required by the algorithm to perform a given task as a function of the *size* or *length* of the task. Similarly, the *space complexity* of an algorithm quantifies the amount of storage (memory) required by the algorithm as a function of the task size/length. We are typically mainly interested in the asymptotic behavior rather than exact quantities. To this end, the so-called "Big O" notation is commonly used.

1 Big O notation

A function $f(x)$ is said to be "big o" of $g(x)$, written as $f(x) = O(g(x))$ as $x \to \infty$, if there exists positive constants c and x_0 such that

$$|f(x)| \leq c\, g(x), \quad \forall x \geq x_0.$$

In other words, the function g can be used to construct an upper bound on the *worst-case* asymptotic behavior of f.

As an example, suppose $f(x) = 4x^3 - 2x^2 + 5x$. Asymptotically, the first term (i.e., $4x^3$) grows much faster than the last two terms, so $f(x) = O(x^3)$ as $x \to \infty$. Indeed, we have that

$$|f(x)| \leq |4x^3| + |2x^2| + |5x|$$

and hence

$$|f(x)| \leq 11x^3, \quad \forall x \geq 1.$$

Note that $f(n) = O(x^4)$ and $f(n) = O(2^n)$ are also true if $f(x) = O(x^3)$. In practice, we are often interested in the *least conservative* bound on the asymptotic growth rate of f. A lower bound on the asymptotic behavior of f leads the the so-called *big omega* notation: $f(x)$ is said to be "big omega" of $g(x)$, written as $f(x) = \Omega(g(x))$ as $x \to \infty$, if $g(x) = O(f(x))$, i.e.,

$$f(x) = \Omega(g(x)) \iff g(x) = O(f(x)).$$

(We note that are more general definition of exists.) If $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$, then $f(x)$ is said to be "big theta" of $g(x)$, written as $f(x) = \Theta(g(x))$ as $x \to \infty$. In other words, $f(x)$ is bounded above and below asymptotically by a scalar multiple of $g(x)$.

Now suppose that $f(n)$ represents the time required by some algorithm or program as a function of the input size n. The time complexity of the algorithm may then be classified according to the asymptotic behavoir of f. For example, the time complexity is said to be polynomial if $f(n) = O(n^k)$ for some positive integer k, and it is exponential if $f(n) = O(2^{\text{poly}(n)})$ where poly(n) denotes a polynomial in n. The following table lists a number of complexity classes along with their Big O characterization.

Complexity	Big O notation
Constant	$O(1)$
Logarithmic	$O(\log n)$
Poly-logarithmic	$O((\log n)^k)$
Linear	$O(n)$
Quasi-linear/log-linear	$O(n(\log n)^k)$
Polynomial	$O(n^k)$
Exponential	$O(2^{\mathrm{poly}(n)})$
Factorial	$O(n!)$
Double exponential	$O(2^{2^{\mathrm{poly}(n)}})$

The example in the following table illustrates the difference between polynomial and exponential time complexity. Clearly, exponential time complexity grows much faster than polynomial time complexity.

$f(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 40$
n	0.00001 s	0.00002 s	0.00003 s	0.00004 s
n^2	0.0001 s	0.0004 s	0.0009 s	0.0016 s
n^3	0.001 s	0.008 s	0.027 s	0.064 s
n^5	0.1 s	3.2 s	24.3 s	1.7 min
2^n	0.001 s	1 s	17.9 min	12.7 days
3^n	0.059 s	58 min	6.5 years	3855 centuries

Example 1

The travelling salesman problem is a combinatorial problem that seeks the shortest (or cheapest) route through n locations: the route must visit each location exactly once and return to the starting point (one of the n locations). There are $n!$ possible routes, so an algorithm that finds the shortest route by comparing all of these will have *factorial* time complexity.

Example 2

The Fibonacci sequence is defined recursively as

$$f(n) = f(n-1) + f(n-2), \quad n \geq 2,$$

where $f(0) = 0$ and $f(1) = 1$. The sequence is monotonically increasing which implies that

$$f(n) = f(n-1) + f(n-2) \leq 2f(n-1), \quad n \geq 2,$$

since $f(n-1) \geq f(n-2)$. In other words, $f(n)$ at most twice the value of $f(n-1)$ (provided that $n \geq 2$), and hence $f(n) = O(2^n)$. Monotonicity also implies that

$$f(n) = f(n-1) + f(n-2) \geq 2f(n-2), \quad n \geq 2,$$

which, in turn, implies that $f(n) = \Omega(2^{n/2})$ since the Fibonacci sequence at least doubles every time we increase n by 2. Consequently, we can conclude that, asymptotically, the growth of $f(n)$ is exponential.

2 Floating-point operations

The number of floating-point operations or *FLOPs* (floating-point addition, subtraction, multiplication, and division) is often used as a rough measure of the time complexity of matrix and vector operations. For example, the inner product of two length n vectors x and y, defined as

$$x^T y = \sum_{k=1}^{n} x_i y_i,$$

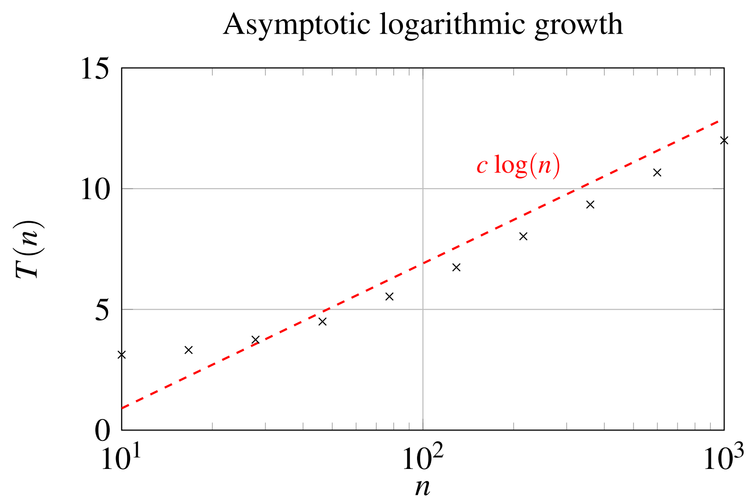
requires $2n-1$ FLOPs (n scalar multiplications and $n-1$ additions), and the matrix-vector product $y = Ax$ where A is $m \times n$ is equivalent to m such inner products (i.e., $y_i = a_i^T x$ where a_i^T denotes the ith row of A), and hence it requires $m(2n-1)$ FLOPs. Thus, assuming that one FLOP requires a constant amount of time, the time complexity of an inner product is $O(n)$ whereas the time complexity of matrix-vector multiplication is $O(mn)$. In the special case where A is a square matrix (i.e., $m = n$), the time complexity of the matrix-vector product is $O(n^2)$.

We note that "FLOP" refers to a single floating-point operation, the plural of which is "FLOPs". The abbreviation "FLOPS" (with uppercase "S") generally means *floating-point operations per second* and is used as a measure of performance.

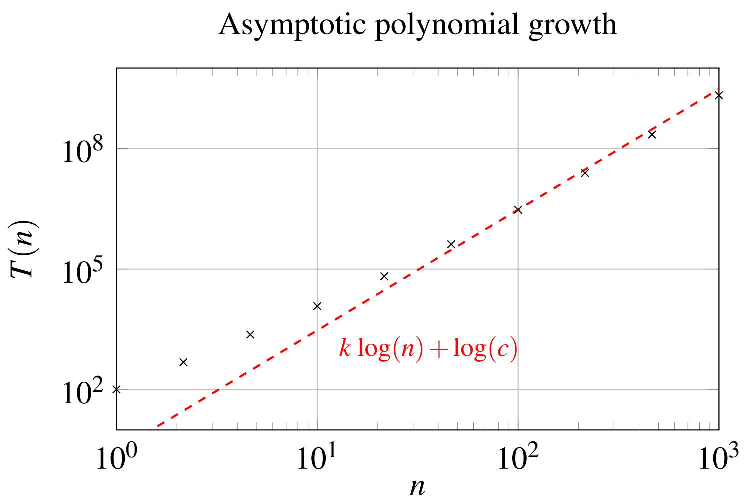
3 Complexity plots

Suppose we have measured the computation time of some algorithm for k different problem sizes $n_1, n_2, \ldots, n_k$. We will let T_i denote our measurement for problem size n_i. The question is now as follows: how can we assess the growth rate visually by plotting our measurements?

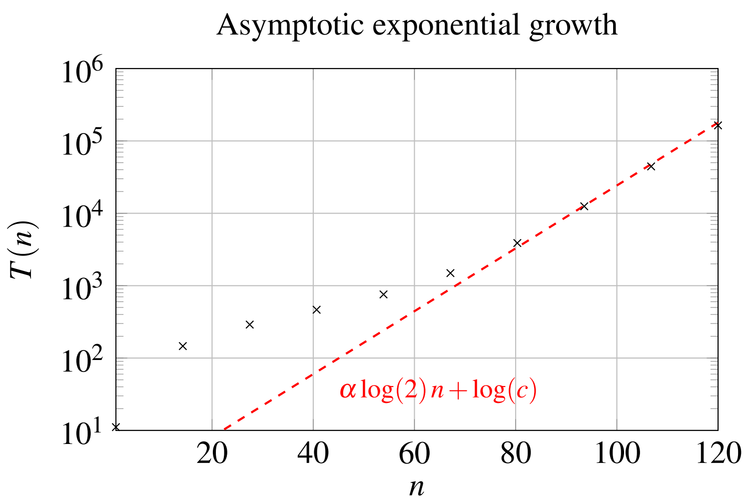
Suppose our hypothesis is that the computation time grows at most logarithmically as a function of n, i.e., $T(n) = O(\log(n))$. In other words, we should be able to find a positive constant c such that $T(n) \leq c\log(n)$ holds asymptotically. Thus, visualizing our measurements in a semilogarithmic plot with a logarithmic n-axis should expose an asymptotic growth that is at most linear in $\log(n)$. The following figure illustrates this.



Next, suppose our hypothesis is that the computation time is upper bounded by a polynomial in n, i.e., $T(n) = O(n^k)$ for some integer $k \geq 1$. We may then plot our measurements (n_i, T_i) using a logarithmic scale for both axes, which should expose an asymptotic growth that satisfies $\log(T(n)) \leq k\log(n) + \log(c)$ for some positive constant c. This illustrated in the next figure.



As a last example, suppose our hypothesis is that the computation time is upper bounded by an exponential function, i.e., $T(n) = O(2^{\alpha n})$ for some positive constant α. Plot our measurements (n_i, T_i) using a logarithmic T-axis turns an exponential function into a line, i.e., the asymptotic upper bound becomes $\log(T(n)) \le \alpha \log(2)\, n + \log(c)$. This is illustrated in the next figure.



The following table summarizes how the time complexity can be analyzed by means of plotting measurements for a number of different problem sizes.

Hypothesis	Plot	Upper bound	Slope
$O(\log(n))$	Log n-axis	$T(n) \le c \log(n)$	c
$O(n^k)$	Log-log plot	$\log(T(n)) \le k \log(n) + \log(c)$	k
$O(2^{\alpha n})$	Log T-axis	$\log(T(n)) \le \alpha \log(2)\, n + \log(c)$	$\alpha \log(2)$