Growth-rate Functions

- O(1) – **constant** time, the time is independent of *n*, e.g. array look-up
- O(log *n*) – **logarithmic** time, usually the log is base 2, e.g. binary search
- O(*n*) – **linear** time, e.g. linear search
- O(*n**log *n*) – e.g. efficient sorting algorithms
- O(n^2) – **quadratic** time, e.g. selection sort
- O(n^k) – **polynomial** (where *k* is some constant)
- O(2^n) – **exponential** time, very slow!

- Order of growth of some common functions

  $O(1) < O(\log n) < O(n) < O(n * \log n) < O(n^2) < O(n^3) < O(2^n)$

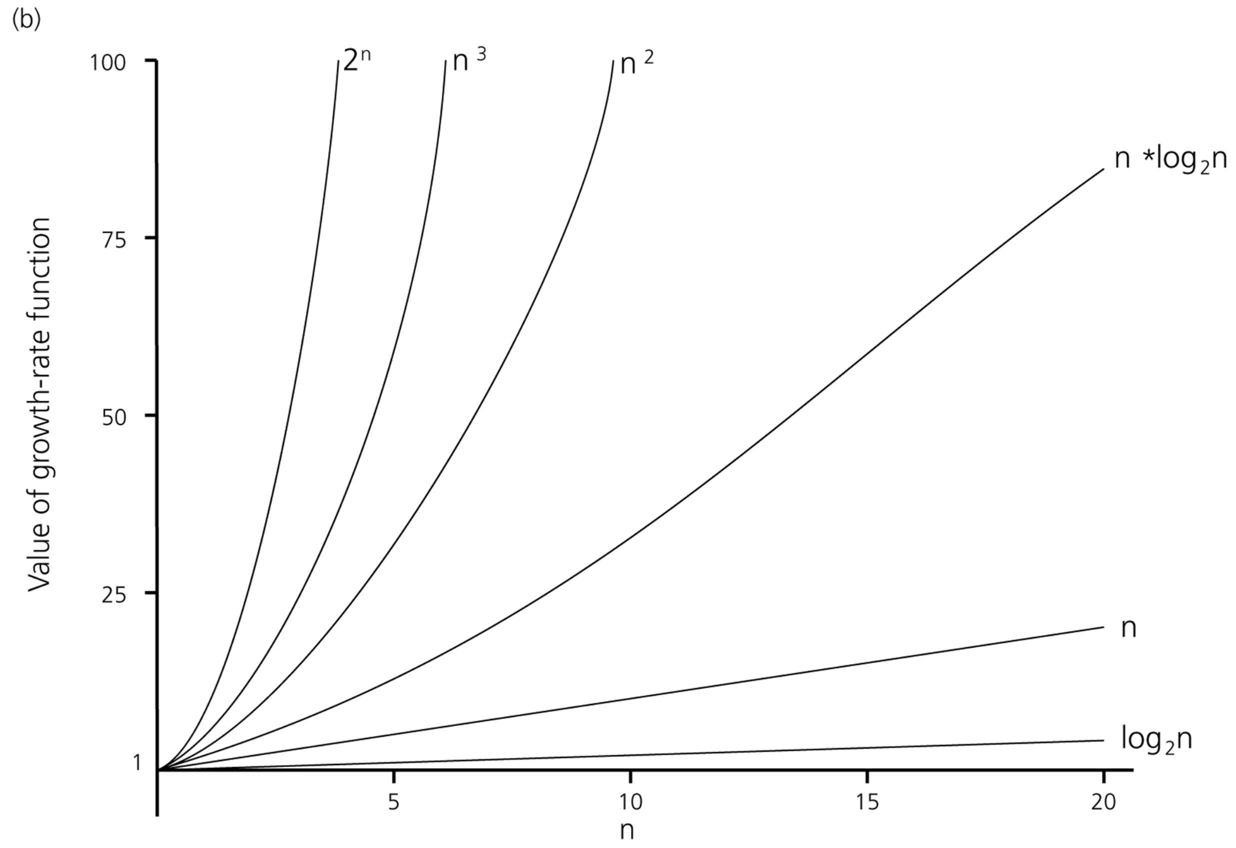
Order-of-Magnitude Analysis and Big O Notation

(a)

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

A comparison of growth-rate functions: a) in tabular form

Order-of-Magnitude Analysis and Big O Notation



A comparison of growth-rate functions: b) in graphical form

Note on Constant Time

- We write O(1) to indicate something that takes a constant amount of time
  - E.g. finding the minimum element of an ordered array takes O(1) time, because the min is either at the beginning or the end of the array
  - **Important**: constants can be huge, and so in practice O(1) is not necessarily efficient --- all it tells us is that the algorithm will run at the same speed no matter the size of the input we give it

Arithmetic of Big-O Notation

1) If $f(n)$ is $O(g(n))$ then $c.f(n)$ is $O(g(n))$, where c is a constant.
   - Example: $23*\log n$ *is* $O(\log n)$

2) If $f_1(n)$ is $O(g(n))$ and $f_2(n)$ is $O(g(n))$ then also $f_1(n)+f_2(n)$ is $O(g(n))$
   - Example: what is order of n^2+n?
     n^2 is $O(n^2)$
     n is $O(n)$ but also $O(n^2)$
     therefore n^2+n is $O(n^2)$

Arithmetic of Big-O Notation

3) If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n)*f_2(n)$ is $O(g_1(n)*g_2(n))$.

- Example: what is order of (3n+1)*(2n+log n)?
  3n+1 is O(n)
  2n+log n is O(n)
  (3n+1)*(2n+log n) is $O(n*n)=O(n^2)$

Using Big O Notation

- It's not correct to say:
  $f(n) \leq O(g(n))$,
  $f(n) = O(g(n))$
- It's completely wrong to say:
  $f(n) \geq O(g(n))$
  $f(n) > O(g(n))$
- Just use:
  $f(n)$ is (in) $O(g(n))$, or
  $f(n)$ is of order $O(g(n))$, or
  $f(n) \in O(g(n))$

Using Big O Notation

- Sometimes we need to be more specific when comparing the algorithms.
- For instance, there might be several sorting algorithms with time of order O(n.log n). However, an algorithm with cost function
  2n.log n + 10n + 7log n + 40
  is better than one with cost function
  5n.log n + 2n +10log n +1
- That means:
  - We care about the constant of the main term.
  - But we still don't care about other terms.
- In such situations, the following notation is often used:
  2n.log n + O(n) for the first algorithm
  5n.log n + O(n) for the second one

Searching costs using O-notation

- Linear search
  - Best case: O(1)
  - Average case: O(n)
  - Worst case: O(n)
- Binary search
  - Best case: O(1)
  - Average case: O(log n)
  - Worst case: O(log n)

Sorting cost in O-notation

- Selection sort
  - Best case: $O(n^2)$ (can vary with implementation)
  - Average case: $O(n^2)$
  - Worst case: $O(n^2)$
- Insertion sort
  - Best case: $O(n)$ (can vary with implementation)
  - Average case: $O(n^2)$
  - Worst case: $O(n^2)$