Design and Analysis of Algorithms

Part 1

Program Cost and Asymptotic Notation

Elias Koutsoupias
with thanks to Giulio Chiribella

Hilary Term 2021

Fast computers *vs* efficient algorithms [CLRS 1]

Many recent innovations rely on

- fast computers
- efficient algorithms.

Which is more important?

The importance of efficient algorithms

The *cost* of an algorithm can be quantified by the number of steps $T(n)$ in which the algorithm solves a problem of size n.

Imagine that a certain problem can be solved by four different algorithms, with $T(n) = n, n^2, n^3$, and 2^n, respectively.
Question: what is the maximum problem size that the algorithm can solve in a given time?
Assume that a computer is capable of 10^{10} steps per second.

Cost $T(n)$ (Complexity)	Maximum problem size solvable in		
	1 second	1 hour	1 year
n	10^{10}	3.6×10^{13}	3×10^{17}
n^2	10^5	6×10^6	5×10^8
n^3	2154	33000	680000
2^n	33	45	58

Faster computers *vs* more efficient algorithms

Suppose a faster computer is capable of 10^{16} steps per second.

Cost $T(n)$	Max. size before	Max. size now
n	s_1	$10^6 \times s_1$
n^2	s_2	$1000 \times s_2$
n^3	s_3	$100 \times s_3$
2^n	s_4	$s_4 + 20$

A $10^6\times$ increase in speed results in only a factor-of-100 improvement if cost is n^3, and only an additive increase of 20 if cost is 2^n.

Conclusions As computer speeds increase ...

1. ... it is algorithmic efficiency that really determines the increase in problem size that can be achieved.
2. ... so does the size of problems we wish to solve.
   Thus, designing efficient algorithms becomes even more important!

From *Algorism* to *Algorithms*

Invented in India around AD 600, the *decimal system* was a revolution in quantitative reasoning. Arabic mathematicians helped develop arithmetic methods using the Indian decimals.

A 9th-century Arabic textbook by the Persian *Al Khwarizmi* was the key to the spread of the Indian-Arabic decimal arithmetic. He gave methods for basic arithmetic (adding, multiplying and dividing numbers), even the calculation of square roots and digits of π.

Derived from 'Al Khwarizmi', *algorism* means rules for performing arithmetic computations using the Indian-Arabic decimal system.

The word "algorism" devolved into *algorithm*, with a generalisation of the meaning to

Algorithm: a finite set of well-defined instructions for accomplishing some task.

Evaluating algorithms

Two questions we ask about an algorithm

1. Is it correct?
2. Is it efficient?

Correctness is of utmost importance.
It is easy to design a highly efficient but incorrect algorithm.

Efficiency with respect to:

- Running time
- Space (amount of memory used)
- Network traffic
- Other features (e.g. number of times secondary storage is accessed)

Proving correctness and analysing the efficiency of programs are difficult problems, in general. Take for example the Collatz program: starting from a positive integer x repeat “if x is even then $x = x/2$, else $x = (3x + 1)/2$” until $x = 1$. We don’t know how many steps this program takes.

Measuring running time

On an actual computer, the running time of a *program* depends on many factors:

1. The running time of the algorithm.
2. The input of the program.
3. The quality of the implementation (e.g. quality of the code generated by the compiler).
4. The machine running the program.

We are concerned with 1.

Sorting [CLRS 2.1]

The Sorting Problem

Input: A sequence of n integers $a_1, \cdots, a_n$.

Output: A permutation $a_{\sigma(1)}, \cdots, a_{\sigma(n)}$ of the input such that

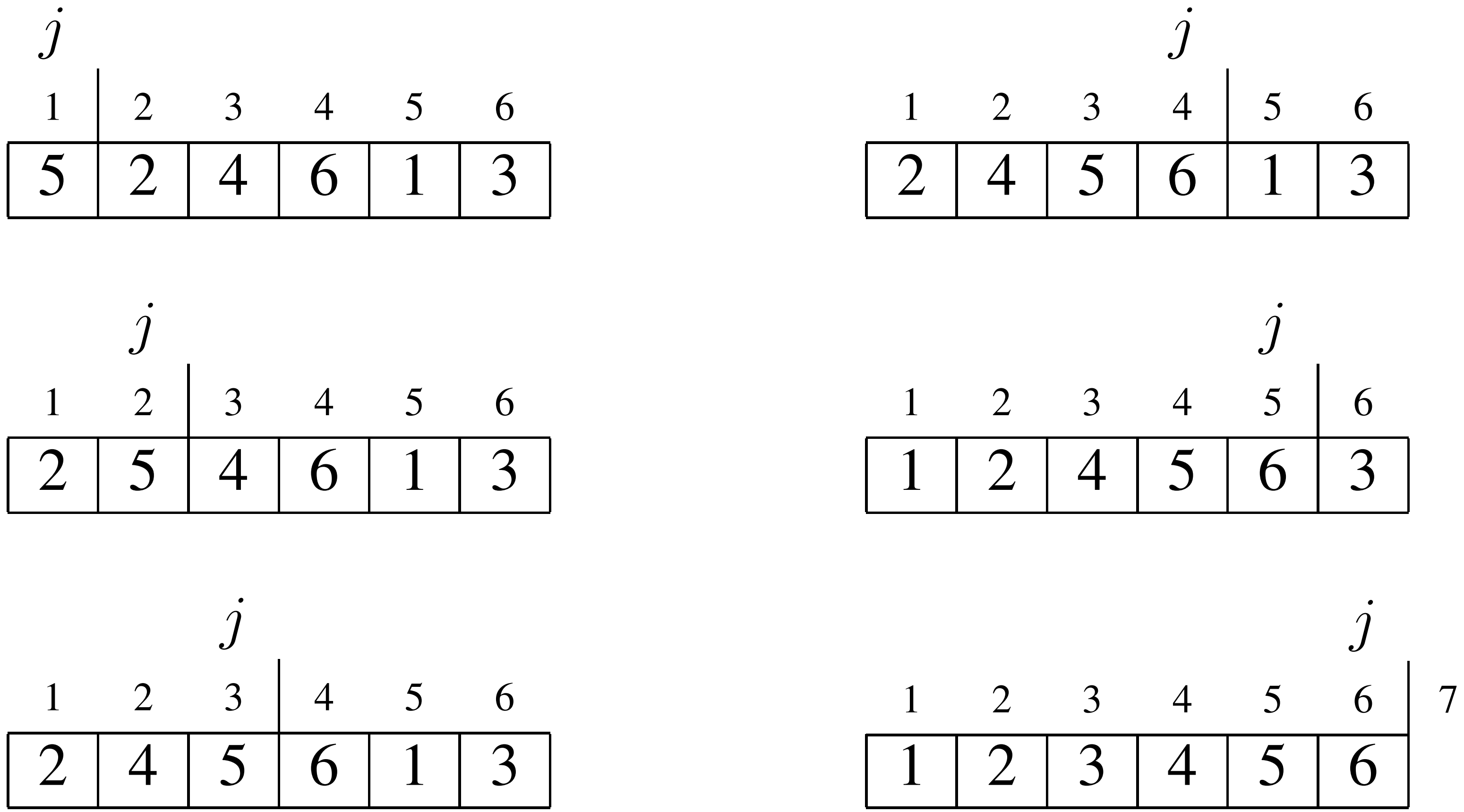
$$a_{\sigma(1)} \leq a_{\sigma(2)} \leq \cdots \leq a_{\sigma(n)}.$$

The sequences are typically stored in arrays.

Insertion sort: Informal description

- The input is an integer array $A[1\,..\,n]$,
  with $A[1] = a_1, A[2] = a_2, \ldots, A[n] = a_n$.
- The algorithm consists of $n - 1$ iterations.
  At the j-th iteration,
  the first $j + 1$ entries of the array $A[1\,..\,n]$ are arranged in sorted order.
  To do this, the entry $A[j + 1]$ is compared with the entry $A[i]$, $i \leq j$,
  starting from $i = j$.
  - If $A[i] > A[j + 1]$, the value $A[i]$ is moved from the i-th position to the $(i + 1)$-th position, and the counter i is decremented to $i - 1$.
  - If $A[i] \leq A[j + 1]$, the value $A[j + 1]$ is put into the $(i + 1)$-th position of the array and the iteration terminates.

Example



Observation. At the start of the j-th iteration, the subarray $A[1\,..\,j]$ consists of the elements originally in $A[1\,..\,j]$ but in sorted order.

Pseudocode (CRLS-style)

INSERTION-SORT(A)

Input: An integer array A
Output: Array A sorted in non-decreasing order

```
for j = 1 to A.length − 1
    key = A[j + 1]
    // Insert A[j + 1] into the sorted sequence A[1 . . j].
    i = j
    while i > 0 and A[i] > key
        A[i + 1] = A[i] // moves the value A[i] one place to the right
        i = i − 1
    A[i + 1] = key
```

Characteristics of the CLRS pseudocode

- Similar to Pascal, C and Java.
- Pseudocode is for communicating algorithms to humans: many programming issues (e.g. data abstraction, modularity, error handling, etc.) are ignored.
- English statements are sometimes used.
- "//" indicates that the reminder of the line is a comment. (In 2nd edition "$\triangleright$" is used.)
- Variables are local to the block, unless otherwise specified.
- Block structure is indicated by indentation.
- Assignment "$x = y$" makes x reference the same object as y. (In 2nd edition "$\leftarrow$" is used.)
- Boolean operators "and" and "or" are "short-circuiting".

Loop-invariant approach to correctness proof

Three key components of a loop-invariant argument:

1. *Initialization*: Prove that invariant (I) holds prior to first iteration.
2. *Maintenance*: Prove that if (I) holds just before an iteration, then it holds just before the next iteration.
3. *Termination*: Prove that when the loop terminates, the invariant (I), *and* the reason that the loop terminates, imply the correctness of the loop-construct.

The approach is reminiscent of mathematical induction:

1. Initialization corresponds to establishing the base case.
2. Maintenance corresponds to establishing the inductive case.
3. The difference is that we expect to exit the loop,
   whereas mathematical induction establishes a result for all natural numbers.

Correctness of INSERTION-SORT

Invariant of outer loop: At the start of the j-th iteration, the subarray $A[1 \mathinner{.\,.} j]$ consists of the elements originally in $A[1 \mathinner{.\,.} j]$ but in sorted order.

Correctness of Insertion-Sort

Invariant of outer loop: At the start of the j-th iteration, the subarray $A[1\,..\,j]$ consists of the elements originally in $A[1\,..\,j]$ but in sorted order.

Initialization. When $j = 1$, the subarray $A[1\,..\,j]$ is a singleton and trivially sorted.

Termination. The outer **for** loop terminates when $j = n := A.length$. With $j = n$, the invariant reads: $A[1\,..\,n]$ consists of the elements originally in $A[1\,..\,n]$ but in sorted order.

Correctness of INSERTION-SORT

Invariant of outer loop: At the start of the j-th iteration, the subarray $A[1\,..\,j]$ consists of the elements originally in $A[1\,..\,j]$ but in sorted order.

Initialization. When $j = 1$, the subarray $A[1\,..\,j]$ is a singleton and trivially sorted.

Termination. The outer **for** loop terminates when $j = n := A.length$. With $j = n$, the invariant reads: $A[1\,..\,n]$ consists of the elements originally in $A[1\,..\,n]$ but in sorted order.

Maintenance. Suppose input is sequence $a_1, \cdots, a_n$.
We need to prove the following:

If at the start of the j-th iteration $A[1\,..\,j]$ consists of $a_1, \cdots, a_j$ in sorted order, then at the start of the $(j+1)$-th iteration $A[1\,..\,j+1]$ consists of $a_1, \cdots, a_{j+1}$ in sorted order.

The proof requires us to examine the behaviour of the inner **while** loop, under the promise that the subarray $A[1\,..\,j]$ consists of $a_1, \cdots, a_j$ in sorted order.

Correctness of INSERTION-SORT, continued

The inner **while** loop has the following property:

Property of the while loop: if $A[1\,..\,j]$ consists of $a_1, \cdots, a_j$ in sorted order, then, at termination of the **while** loop, the sequence $A[1\,..\,i]\; key\; A[i{+}2\,..\,j{+}1]$ consists of $a_1, \cdots, a_{j+1}$ in sorted order.

This property implies the maintenance of the loop invariant of the **for** loop, because the array $A[1\,..\,j+1]$ after the j-th iteration of the **for** loop is exactly the sequence $A[1\,..\,i]\; key\; A[i{+}2\,..\,j{+}1]$.

Correctness of INSERTION-SORT, continued

The inner **while** loop has the following property:

Property of the while loop: if $A[1\,..\,j]$ consists of $a_1, \cdots, a_j$ in sorted order, then, at termination of the **while** loop, the sequence $A[1\,..\,i]\ key\ A[i{+}2\,..\,j{+}1]$ consists of $a_1, \cdots, a_{j+1}$ in sorted order.

This property implies the maintenance of the loop invariant of the **for** loop, because the array $A[1\,..\,j+1]$ after the j-th iteration of the **for** loop is exactly the sequence $A[1\,..\,i]\ key\ A[i{+}2\,..\,j{+}1]$.

Hence, it only remains to prove the validity of the above property.
The proof, provided in the next slide, uses a loop invariant for the **while** loop.

Correctness of INSERTION-SORT, concluded

Invariant of while loop:
(I1) $A[1\,..\,i]\;A[i{+}2\,..\,j{+}1]$ is $a_1, \cdots a_j$ in sorted order
(I2) all elements in $A[i{+}2\,..\,j{+}1]$ are strictly greater than key.

Initialization. For $i = j$, (I1) is true because $A[1,..\,j]$ was promised to be $a_1, \cdots, a_j$ in sorted order, and (I2) is trivially true because the array $A[j + 2\,..\,j + 1]$ is empty.
Termination. Termination occurs if either $i = 0$ or $A[i] \leq key$. In both cases, (I1) and (I2) guarantee that the sequence $A[1\,..\,i]\; key\; A[i{+}2\,..\,j{+}1]$ contains the same elements as $a_1, \cdots a_{j+1}$ in sorted order.
Maintenance. For a given i, the body of the loop is executed only if $A[i] > key$. In that case, $A[i + 1]$ gets the value $A[i]$. After this change, the sequence $A[1\,..\,i - 1]\;A[i{+}1\,..\,j + 1]$ is $a_1, \cdots a_j$ in sorted order, and all elements in $A[i{+}1\,..\,j{+}1]$ are strictly greater than key.
Hence, (I1) and (I2) still hold when i is decremented to $i - 1$.

Running time analysis [CLRS 2.2]

Running time is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of time statement executed})$$

CLRS assume the following model: for a given pseudocode

- □ Line i takes constant time c_i.
- □ When a **for** or **while** loop exits normally,
  the test is executed *one more time* than the loop body.

This model is well justified when each line of the pseudocode contains:

- □ a constant number of basic arithmetic operations
  (add, subtract, multiply, divide, remainder, floor, ceiling)
- □ a constant number of data movement instructions (load, store, copy)
- □ a constant number of control instructions
  (conditional and unconditional branch, subroutine call and return)

The running time of INSERTION-SORT

Recall the pseudocode:

```
for j = 1 to A.length − 1
    key = A[j + 1]
    // Insert A[j + 1] into the sorted sequence A[1 . . j].
    i = j
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Setting $n := A.length$, the running time is

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \sum_{j=1}^{n-1} t_j + c_6 \sum_{j=1}^{n-1}(t_j - 1) + c_7 \sum_{j=1}^{n-1}(t_j - 1) + c_8(n-1),$$

where t_j is the number of times the test of the **while** loop is executed for a given value of j (note that t_j may also depend on the input).

Worst-case analysis

- The input array contains distinct elements in reverse sorted order i.e. is strictly decreasing.
- Why? Because we have to compare $key = a_{j+1}$ with every element to left of the $(j+1)$-th element, and so, compare with j elements in total.
- Thus $t_j = j + 1$. We have $\sum_{j=1}^{n-1} t_j = \sum_{j=1}^{n-1} (j+1) = \frac{n(n+1)}{2} - 1$, and so,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(\tfrac{n(n+1)}{2} - 1) \\ &\quad + c_6(\tfrac{n(n-1)}{2}) + c_7 \tfrac{n(n-1)}{2} + c_8(n-1) \\ &= an^2 + bn + c \end{aligned}$$

for appropriate a, b and c.

Hence $T(n)$ is a *quadratic* function of n.

Best-case analysis

- The array is already sorted.
- Always find $A[i] \leq key$ upon the first iteration of the **while** loop (when $i = j$).
- Thus $t_j = 1$.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

I.e. $T(n)$ is a *linear* function of n.

Average-case analysis (informal)

- Randomly choose n numbers as input.
- On average, the key in $A[j]$ is less than half the elements in $A[1\,..\,j]$ and greater than the other half, and so, on average the **while** loop has to look halfway through the sorted subarray $A[1\,..\,j]$ to decide where to drop the key.
- Hence $t_j = j/2$.
- Although average-case running time is approximately half that of the worst-case, it is still quadratic.

Moral. Average-case complexity can be *asymptotically* as bad as the worst-case.

Average-case analysis is not straightforward:

- What is meant by "average input" depends on the application.
- The mathematics can be difficult.

Features of insertion sort: a summary

- Worst-case quadratic time.
- Linear-time on already sorted (or nearly sorted) inputs.

- ***Stable***: Relative order of elements with equal keys is maintained.
- ***In-place***: Only a constant amount of extra memory space (other than that which holds the input) is required, regardless of the size of the input.
- ***Online***: it can sort a list as it is received.

The Big-O notation [CLRS 3]

Let $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$ be functions. Define the set

$$O(g(n)) \;:=\; \{\, f : \mathbb{N} \longrightarrow \mathbb{R}^+ : \exists n_0 \in \mathbb{N}^+ \,.\, \exists c \in \mathbb{R}^+ \,.\, \forall n \,. \\ n \geq n_0 \;\rightarrow\; f(n) \leq c \cdot g(n) \,\}$$

In words, $f \in O(g)$ if there exist a positive integer n_0 and a positive real c such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Informally $O(g)$ is the set of functions that are bounded above by g, ignoring constant factors, and ignoring a finite number of exceptions.

If $f \in O(g)$, then we say that "g is an ***asymptotic upper bound*** for f"

Examples

$$O(g(n)) \;:=\; \{\, f : \mathbb{N} \longrightarrow \mathbb{R}^+ : \exists n_0 \in \mathbb{N}^+ \,.\, \exists c \in \mathbb{R}^+ \,.\, \forall n \,.\, n \geq n_0 \;\rightarrow\; f(n) \leq c \cdot g(n) \,\}$$

1. $3^{98} \in O(1)$ [regarding 3^{98} and 1 as (constant) functions of n].
   Take $n_0 = 1$ and $c = 3^{98}$.
2. $5n^2 + 9 \in O(n^2)$.
   Take $n_0 = 3$ and $c = 6$. Then for for all $n \geq n_0$, we have $9 \leq n^2$, and so $5n^2 + 9 \leq 5n^2 + n^2 = 6n^2 = cn^2$.
3. Take $g(n) = n^2$ and $f(n) = 7n^2 + 3n + 11$. Then $f \in O(g)$.
4. Some more functions in $O(n^2)$:
   $1000n^2$, n, $n^{1.9999}$, $n^2/\lg\lg\lg n$ and 6.

Properties of Big-O

Lemma 1. *Let* $f, g, h : \mathbb{N} \longrightarrow \mathbb{R}^+$. *Then:*

1. *For every constant* $c > 0$*, if* $f \in O(g)$ *then* $c\,f \in O(g)$.
2. *For every constant* $c > 0$*, if* $f \in O(g)$ *then* $f \in O(c\,g)$.
3. *If* $f_1 \in O(g_1)$ *and* $f_2 \in O(g_2)$ *then* $f_1 + f_2 \in O(g_1 + g_2)$.
4. *If* $f_1 \in O(g_1)$ *and* $f_2 \in O(g_2)$ *then* $f_1 + f_2 \in O(\max(g_1, g_2))$.
5. *If* $f_1 \in O(g_1)$ *and* $f_2 \in O(g_2)$ *then* $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$.
6. *If* $f \in O(g)$ *and* $g \in O(h)$ *then* $f \in O(h)$.
7. *Every polynomial of degree* $l \geq 0$ *is in* $O(n^l)$.
8. *For any* $c > 0$ *in* $\mathbb{R}$*, we have* $\lg(n^c) \in O(\lg(n))$.
9. *For every constant* $c, d > 0$*, we have* $\lg^c(n) \in O(n^d)$.
10. *For every constant* $c > 0$ *and* $d > 1$*, we have* $n^c \in O(d^n)$.
11. *For every constant* $0 \leq c \leq d$*, we have* $n^c \in O(n^d)$.

Example. Show that

$$57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 \;\in\; O(n^3)$$

by appealing to Lemma 1.

$$\begin{array}{rcll}
\lg^5(n) & \in & O(n) & \because 9 \\
4n^2 \cdot \lg^5(n) & \in & O(4n^3) & \because 5 \\
57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 & \in & O(57n^3 + 4n^3 + 17n + 498) & \because 3 \\
57n^3 + 4n^3 + 17n + 498 & \in & O(n^3) & \because 7 \\
57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 & \in & O(n^3) & \because 6
\end{array}$$

A shorthand for Big-O

Instead of writing $f \in O(g)$ we often write

$$f(n) \ = \ O(g(n))$$

(read "f is Big-O of g").

A shorthand for Big-O

Instead of writing $f \in O(g)$ we often write

$$f(n) \ = \ O(g(n))$$

(read "f is Big-O of g").

It is also convenient to write

$$f_1(n) \ = \ f_2(n) + O(g(n))$$

meaning that

$$f_1(n) = f_2(n) + h(n)\,,$$

where $h(n)$ is a generic function in $O(g(n))$.

Pitfalls of the shorthand

When writing

$$f(n) \ = \ O(g(n))$$

we must bear in mind that it is a shorthand for $f(n) \in O(g(n))$.

Here "=" is **not** an equality between two objects.
In particular, it does **not** have the transitive property:
$f(n) = \ O(g(n))$ and $h(n) = \ O(g(n))$ does **not** imply $f(n) = h(n)$!

Two elements of the same set are not necessarily the same element!

Example

□ $n = O(n^3)$ and $n^2 = O(n^3)$ but $n \neq n^2$.

So why use the shorthand?

- It is convenient to write equations like $f(n) = g(n) + O(n^d)$, or $f(n) = g(n)\,(1 + O(1/n))$
- The Big-O shorthand is a very common mathematical notation, in use since more than 100 years ago.
- We already abuse the "$=$" symbol in computer science: think of the pseudocode instruction $i = i - 1$.

Big-Ω

The Big-O notation is useful for upper bounds. There is an analogous notation for lower bounds, called the Big-Omega. We write

$$f(n) = \Omega(g(n))$$

to mean "there exist a positive integer n_0 and a positive real c such that for all $n \geq n_0$, we have $f(n) \geq cg(n)$."

If $f \in \Omega(g)$, then we say that "g is an ***asymptotic lower bound*** for f".

Example

1. $n^n = \Omega(n!)$
2. $2^n = \Omega(n^{10})$.

Exercise

Prove that $f(n) = \Omega(g(n))$ iff $g(n) = O(f(n))$.

Big-Θ

When the function $g(n)$ is both an asymptotic upper bound and an asymptotic lower bound for $f(n)$, we say that "g is an ***asymptotic tight bound*** for f", and we write

$$f(n) = \Theta(g(n))\,.$$

Equivalently, $f = \Theta(g)$ means that there exist positive reals c_1 and c_2 and a positive integer n_0 such that for all $n \geq n_0$, we have

$$c_1 g(n) \;\leq\; f(n) \;\leq\; c_2 g(n).$$

We can think of f and g as having the "same order of magnitude".

Examples

1. $5n^3 + 88n = \Theta(n^3)$
2. $2 + \sin(\lg n) = \Theta(1)$
3. $n! = \Theta(n^{n+1/2}e^{-n})$.
   Consequence of Stirling's Approximation.
4. For all $a, b > 1$, $\log_a n = \Theta(\log_b n)$.
   Consequence of the relation $\log_b a = \frac{\log_c a}{\log_c b}$.
   From now on, we will be "neutral" and write $\Theta(\log n)$, without specifying the base of the logarithm.

Examples

1. $5n^3 + 88n = \Theta(n^3)$
2. $2 + \sin(\lg n) = \Theta(1)$
3. $n! = \Theta(n^{n+1/2}e^{-n})$.
   Consequence of Stirling's Approximation.
4. For all $a, b > 1$, $\log_a n = \Theta(\log_b n)$.
   Consequence of the relation $\log_b a = \frac{\log_c a}{\log_c b}$.
   From now on, we will be "neutral" and write $\Theta(\log n)$, without specifying the base of the logarithm.

Question. OK, we have seen that $\log_a n$ is a Big-Theta of $\log_b n$.
But is $2^{\log_a n}$ a Big-Theta of $2^{\log_b n}$?

Examples

1. $5n^3 + 88n = \Theta(n^3)$
2. $2 + \sin(\lg n) = \Theta(1)$
3. $n! = \Theta(n^{n+1/2}e^{-n})$.
   Consequence of Stirling's Approximation.
4. For all $a, b > 1$, $\log_a n = \Theta(\log_b n)$.
   Consequence of the relation $\log_b a = \frac{\log_c a}{\log_c b}$.
   From now on, we will be "neutral" and write $\Theta(\log n)$, without specifying the base of the logarithm.

Question. OK, we have seen that $\log_a n$ is a Big-Theta of $\log_b n$. But is $2^{\log_a n}$ a Big-Theta of $2^{\log_b n}$?

No! Recall that $a^{\log_b c} = c^{\log_b a}$ for all $a, b, c > 0$.
Using this relation, we have $2^{\log_a n} = n^{\log_a 2}$ and $2^{\log_b n} = n^{\log_b 2}$.
If $a \neq b$, we have two different powers of n, but $n^c = \Theta(n^d)$ only if $c = d$.

Revision

Logarithms.

$\log_2 n$ is sometimes written $\lg n$, and $\log_e n$ is sometimes written $\ln n$.

Recall the following useful facts. Let $a, b, c > 0$.

$$
\begin{aligned}
a &= b^{\log_b a} \\
\log_b a &= \frac{\log_c a}{\log_c b} \\
\log_b a &= \frac{1}{\log_a b} \\
a^{\log_b c} &= c^{\log_b a}
\end{aligned}
$$

A form of Stirling's approximation.

$$
n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)
$$